(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 126 368 A2

# (12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
**22.08.2001 Bulletin 2001/34**

(51) Int Cl.7: **G06F 9/34**, G06F 9/312

(21) Application number: **01000022.2**

(22) Date of filing: **19.02.2001**

(84) Designated Contracting States:
**AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE TR**
Designated Extension States:
**AL LT LV MK RO SI**

(30) Priority: **18.02.2000 US 183417 P**
**31.10.2000 US 703105**

(71) Applicant: **Texas Instruments Incorporated**
**Dallas, Texas 75251 (US)**

(72) Inventors:
- **Anderson, Timothy D.**
  **Texas 75218, Dallas (US)**
- **Hoyle, David**
  **Arizona 85208, Glendale (US)**
- **Stelss, Donald E.**
  **Texas 75080, Richardson (US)**

(74) Representative: **Holt, Michael**
**Texas Instruments Ltd.,**
**800 Pavilion Drive,**
**Northampton Business Park**
**Northampton, Northamptonshire NN4 7YL (GB)**

## (54) Microprocessor with non-aligned circular addressing

(57) A data processing system (1300) is provided with a digital signal processor (DSP) (1301) that has an instruction set architecture (ISA) that is optimized for intensive numeric algorithm processing. The DSP has dual load/store units (.D1, .D2) connected to dual memory ports (T1, T2) in a level one data cache memory controller (1720a). The DSP can execute two aligned data transfers each having a length of one byte, two bytes, four bytes, or eight bytes in parallel by executing two load/store instructions. The DSP can also execute a single non-aligned data transfer having a length of four bytes or eight bytes by executing a non-aligned load/store instruction that utilizes both memory target ports.

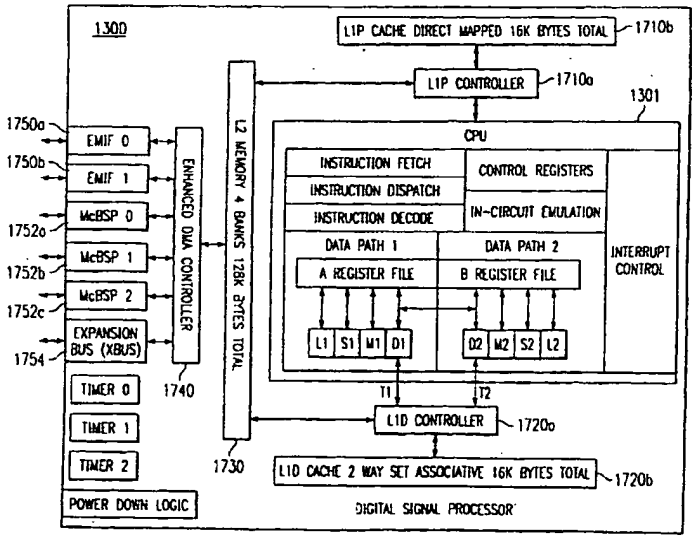



*FIG. 13*

EP 1 126 368 A2

Printed by Jouve, 75001 PARIS (FR)

## Description

### TECHNICAL FIELD OF THE INVENTION

**[0001]** This invention relates to data processing devices, electronic processing and control systems and methods of their manufacture and operation, and particularly relates to memory access schemes of microprocessors optimized for digital signal processing.

### DESCRIPTION OF THE BACKGROUND ART

**[0002]** Generally, a microprocessor is a circuit that combines the instruction-handling, arithmetic, and logical operations of a computer on a single semiconductor integrated circuit. Microprocessors can be grouped into two general classes, namely general-purpose microprocessors and special-purpose microprocessors. General-purpose microprocessors are designed to be programmable by the user to perform any of a wide range of tasks, and are therefore often used as the central processing unit (CPU) in equipment such as personal computers. Special-purpose microprocessors, in contrast, are designed to provide performance improvement for specific predetermined arithmetic and logical functions for which the user intends to use the microprocessor. By knowing the primary function of the microprocessor, the designer can structure the microprocessor architecture in such a manner that the performance of the specific function by the special-purpose microprocessor greatly exceeds the performance of the same function by a general-purpose microprocessor regardless of the program implemented by the user.

**[0003]** One such function that can be performed by a special-purpose microprocessor at a greatly improved rate is digital signal processing. Digital signal processing generally involves the representation, transmission, and manipulation of signals, using numerical techniques and a type of special-purpose microprocessor known as a digital signal processor (DSP). Digital signal processing typically requires the manipulation of large volumes of data, and a digital signal processor is optimized to efficiently perform the intensive computation and memory access operations associated with this data manipulation. For example, computations for performing Fast Fourier Transforms (FFTs) and for implementing digital filters consist to a large degree of repetitive operations such as multiply-and-add and multiple-bit-shift. DSPs can be specifically adapted for these repetitive functions, and provide a substantial performance improvement over general-purpose microprocessors in, for example, real-time applications such as image and speech processing.

**[0004]** DSPs are central to the operation of many of today's electronic products, such as high-speed modems, high-density disk drives, digital cellular phones, complex automotive systems, and video-conferencing equipment. DSPs will enable a wide variety of other digital systems in the future, such as video-phones, network processing, natural speech interfaces, and ultra-high speed modems. The demands placed upon DSPs in these and other applications continue to grow as consumers seek increased performance from their digital products, and as the convergence of the communications, computer and consumer industries creates completely new digital products.

**[0005]** Microprocessor designers have increasingly endeavored to exploit parallelism to improve performance. One parallel architecture that has found application in some modern microprocessors utilizes multiple instruction fetch packets and multiple instruction execution packets with multiple functional units, referred to as a Very Long Instruction Word (VLIW) architecture.

**[0006]** Digital systems designed on a single integrated circuit are referred to as an application specific integrated circuit (ASIC). MegaModules are being used in the design of ASICs to create complex digital systems a single chip. (MegaModule is a trademark of Texas Instruments Incorporated.) Types of MegaModules include SRAMs, FIFOs, register files, RAMs, ROMs, universal asynchronous receiver-transmitters (UARTs), programmable logic arrays and other such logic circuits. MegaModules are usually defined as integrated circuit modules of at least 500 gates in complexity and having a complex ASIC macro function. These MegaModules are predesigned and stored in an ASIC design library. The MegaModules can then be selected by a designer and placed within a certain area on a new IC chip.

**[0007]** Designers have succeeded in increasing the performance of DSPs, and microprocessors in general, by increasing clock speeds, by removing data processing bottlenecks in circuit architecture, by incorporating multiple execution units on a single processor circuit, and by developing optimizing compilers that schedule operations to be executed by the processor in an efficient manner. For example, non-aligned data access is provided on certain microprocessors. Complex instruction set computer (CISC) architectures (Intel, Motorola 68K) have thorough support for non-aligned data accesses; however, reduced instruction set computer (RISC) architectures usually do not support non-aligned accesses with a single load or store instruction. Some RISC architectures allow two data accesses per cycle, but they allow only two aligned accesses. Certain CISC machines now allow doing two memory accesses per cycle as two non-aligned accesses. A reason for this is that the dual access implementations are superscalar implementations that are running code compatible with earlier scalar implementations.

**[0008]** The increasing demands of technology and the marketplace make desirable even further structural and process improvements in processing devices, application systems and methods of operation and manufacture.

## SUMMARY OF THE INVENTION

**[0009]** An examplary embodiment of the present invention seeks to provide a microprocessor and a method for accessing memory by a microprocessor that improves digital signal processing performance. Aspects of the invention are specified in the claims.

**[0010]** In an embodiment of the present invention, each .D unit of a DSP can load and store data items up to double words (up to 64 bits) at aligned addresses in a two port memory subsystem. The .D units can also access words and double words on any byte boundary, whether aligned or not. The address generation circuitry in the first .D unit has a first address output connected to the first memory port and a second address output selectively connected to the second memory port. The address generation circuitry can provide two addresses simultaneously to request two aligned data items. An extraction circuit is connected to the memory subsystem to provide a non-aligned data item extracted from two adjacent aligned data items requested by the .D unit.

**[0011]** In another embodiment of the present invention, the address generation circuitry in the .D units is operable to form an address for non-aligned double word instructions by combining a base address value and an offset value.

**[0012]** In another embodiment of the invention, one or more additional .D units have similar addressing circuitry for non-aligned accesses.

**[0013]** In another embodiment of the present invention, two .D units can simultaneously access aligned data items in the memory.

**[0014]** In another embodiment of the present invention, the memory subsystem has a plurality of memory banks connected to the extraction circuitry. Decode circuitry is connected to the first memory port and to the second memory port for receiving addresses. A plurality of address multiplexers are connected respectively to the plurality of memory banks such that the decode circuitry is operable to individually control each of the plurality of address multiplexers.

## BRIEF DESCRIPTION OF THE DRAWINGS

**[0015]** Other features and advantages of the present invention will become apparent by reference to the following detailed description when considered in conjunction with the accompanying drawings in which the Figures relate to the processor of Figure 1 unless otherwise stated, and in which:

Figure 1 is a block diagram of a digital system with a digital signal processor (DSP), showing components thereof pertinent to an embodiment of the present invention;
Figure 2 is a block diagram of the functional units, data paths and register files of the DSP;
Figure 3A illustrates an opcode map for the load/store instructions which are executed in a .D unit of the DSP;
Figure 3B illustrates an opcode map for the load/store non-aligned double word instruction which are executed in a .D unit of the DSP;
Figure 3C illustrates an opcode map for Boolean instructions executed by the .D units;
Figure 4 illustrates an addressing mode register (AMR) of the DSP;
Figures 5A, 5B and 5C illustrate aspects of non-aligned address formation and non-aligned data extraction from a circular buffer region, according to an aspect of the present invention;
Figure 6 illustrates the basic format of a fetch packet of the DSP;
Figure 7 is a memory map of a portion of the memory space of the DSP and illustrates various aligned and non-aligned memory accesses;
Figure 8 is a block diagram illustrating D-unit address buses of the DSP in more detail and illustrating the two ports of the DSP memory;
Figure 9 is a block diagram of the memory of Figure 8 illustrating address decoding of the two address ports and byte selection circuitry to extract a non-aligned data item according to an embodiment of the present invention;
Figure 10 is a block diagram illustrating the extraction circuitry of Figure 9 in more detail;
Figure 11 is a block diagram illustrating the store byte selection circuitry for storing non-aligned data items in the memory system Figure 8 in more detail;
Figure 12A is a more detailed block diagram of the D-unit of the DSP;
Figure 12B is a more detailed block diagram of the circular buffer circuitry of Figure 12A;
Figure 12C is a flow chart illustrating formation of circular buffer addresses for both aligned access instruction types and non-aligned access instruction types, according to an aspect of the present invention;
Figure 13 is a block diagram of an alternative embodiment of the present invention in digital system having a DSP with a data cache;
Figure 14A and 14B together is a block diagram of the data cache of Figure 13; and
Figure 15 illustrates an exemplary implementation of a digital system that includes an embodiment of the present invention in a mobile telecommunications device.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

**[0016]** Corresponding numerals and symbols in the different figures and tables refer to corresponding parts unless otherwise indicated.

**[0017]** Figure 1 is a block diagram of a microprocessor 1 that has an embodiment of the present invention. Microprocessor 1 is a RISC VLIW digital signal processor ("DSP"). In the interest of clarity, Figure 1 only shows those portions of microprocessor 1 that are relevant to an understanding of an embodiment of the present invention. Details of general construction for DSPs are well known, and may be found readily elsewhere. For example, U.S. Patent 5,072,418 issued to Frederick Boutaud, et al, describes a DSP in detail. U.S. Patent 5,329,471 issued to Gary Swoboda, et al, describes in detail how to test and emulate a DSP. Details of portions of microprocessor 1 relevant to an embodiment of the present invention are explained in sufficient detail hereinbelow, so as to enable one of ordinary skill in the microprocessor art to make and use the invention.

**[0018]** In microprocessor 1 there are shown a central processing unit (CPU) 10, data memory 22, program memory/cache 23, peripherals 60 and an external memory interface (EMIF) with a direct memory access (DMA) 61. CPU 10 further has an instruction fetch/decode unit 10a-c, a plurality of execution units, including an arithmetic and load/store unit D1, a multiplier M1, an ALU/shifter unit S1, an arithmetic logic unit ("ALU") L1, a shared multi-port register file 20a from which data are read and to which data are written. Instructions are fetched by fetch unit 10a from instruction memory 23 over a set of busses 41. Decoded instructions are provided from the instruction fetch/decode unit 10a-c to the functional units D1, M1, S1, and L1 over various sets of control lines which are not shown. Data are provided to/from the register file 20a from/to to load/store units D1 over a first set of busses 32a, to multiplier M1 over a second set of busses 34a, to ALU/shifter unit S1 over a third set of busses 36a and to ALU L1 over a fourth set of busses 38a. Data are provided to/from the memory 22 from/to the load/store units D1 via a fifth set of busses 40a. Note that the entire data path described above is duplicated with register file 20b and execution units D2, M2, S2, and L2. In this embodiment of the present invention, two unrelated aligned double word (64 bits) load/store transfers can be made in parallel between CPU 10 and data memory 22 on each clock cycle using bus set 40a and bus set 40b.

**[0019]** An aligned transfer is one in which the address of a datum modulo its size (in addressible units) is 0. For example, in a system where each 8-bit datum has an address, a 16-bit datum (2 addressible units) at address 0x24002576 is aligned because 0x24002576 mod 2 is 0. Converesly, a non-aligned transfer is one in which the address of a datum modulo its size (in addressible units) is non-zero. For example, in a system where each 8-bit datum has an address, a 32-bit datum (4 addressible units) at address 0x24F78006 is non-aligned because 0x24F78006 mod 4 is 2, not zero.

**[0020]** A single non-aligned double word load/store transfer is performed by scheduling a first .D unit resource and two load/store ports on memory 22. Advantageously, an extraction circuit is connected to the memory subsystem to provide a non-aligned data item extracted from two aligned data items requested by the .D unit. Advantageously, a second .D unit can perform 32-bit logical or arithmetic instructions in addition to the .S and .L units while the address port of the second .D unit is being used to transmit one of two contiguous addresses provided by the first .D unit. Furthermore, a non-aligned access near the end of a circular buffer region in the target memory provides a non-aligned data item that wraps around to the other end of the circular buffer

**[0021]** Emulation circuitry 50 provides access to the internal operation of integrated circuit 1 that can be controlled by an external test/development system (XDS) 51. External test system 51 is representative of a variety of known test systems for debugging and emulating integrated circuits. One such system is described in U.S. Patent 5,535,331. Test circuitry 52 contains control registers and parallel signature analysis circuitry for testing integrated circuit 1.

**[0022]** Note that the memory 22 and memory 23 are shown in Figure 1 to be a part of a microprocessor 1 integrated circuit, the extent of which is represented by the box 42. The memories 22-23 could just as well be external to the microprocessor 1 integrated circuit 42, or part of it could reside on the integrated circuit 42 and part of it be external to the integrated circuit 42. These are matters of design choice. Also, the particular selection and number of execution units are a matter of design choice, and are not critical to the invention.

**[0023]** When microprocessor 1 is incorporated in a data processing system, additional memory or peripherals may be connected to microprocessor 1, as illustrated in Figure 1. For example, Random Access Memory (RAM) 70, a Read Only Memory (ROM) 71 and a Disk 72 are shown connected via an external bus 73. Bus 73 is connected to the External Memory Interface (EMIF) which is part of functional block 61 within microprocessor 1. A Direct Memory Access (DMA) controller is also included within block 61. The DMA controller is generally used to move data between memory and peripherals within microprocessor 1 and memory and peripherals which are external to microprocessor 1.

**[0024]** In the present embodiment, CPU core 10 is encapsulated as a MegaModule, however, other embodiments of the present invention may be in custom designed CPU's or mass market microprocessors, for example.

**[0025]** A detailed description of various architectural features of the microprocessor 1 of Figure 1 is provided in coassigned European Patent Application No.98101291.7 filed 26th January 2000. A description of enhanced architectural features and an extended instruction set not described herein for CPU 10 is provided in coassigned European

Patent Application No.00310098.9 filed 14th November 2000.

**[0026]** Figure 2 is a block diagram of the execution units and register files of the microprocessor of Figure 1 and shows a more detailed view of the buses connecting the various functional blocks. In this figure, all data busses are 32 bits wide, unless otherwise noted. There are two general-purpose register files (A and B) in the processor's data paths. Each of these files contains 32 32-bit registers (A0-A31 for file A and B0-B31 for file B). The general-purpose registers can be used for data, data address pointers, or condition registers. Any number of reads of a given register can be performed in a given cycle.

**[0027]** The general-purpose register files support data ranging in size from packed 8-bit data through 64-bit fixed-point data. Values larger than 32 bits, such as 40-bit long and 64-bit double word quantities, are stored in register pairs, with the 32 LSBs of data placed in an even-numbered register and the remaining 8 or 32 MSBs in the next upper register (which is always an odd-numbered register). Packed data types store either four 8-bit values or two 16-bit values in a single 32-bit register.

**[0028]** There are 32 valid register pairs for 40-bit and 64-bit data, as shown in Table 1. In assembly language syntax, a colon between the register names denotes the register pairs and the odd numbered register is specified first.

**[0029]** Operations requiring a long input ignore the 24 MSBs of the odd register. Operations producing a long result zero-fill the 24 MSBs of the odd register. The even register is encoded in the opcode.

**[0030]** All eight of the functional units have access to the opposite side's register file via a cross path. The .M1, .M2, .S1, .S2, .D1 and .D2 units' src2 inputs are selectable between the cross path and the same side register file by appropriate selection of multiplexers 213, 214 and 215, for example. In the case of the .L1 and .L2 both src1 and src2 inputs are also selectable between the cross path and the same-side register file by appropriate selection of multiplexers 211, 212, for example.

Table 1.

| 40-Bit/64-Bit Register Pairs | |
|---|---|
| Register Files | |
| A | B |
| A1:A0 | B1:B0 |
| A3:A2 | B3:B2 |
| A5:A4 | B5:B4 |
| A7:A6 | 67:B6 |
| A9:A8 | B9:B8 |
| A11:A10 | B11:B10 |
| A13:A12 | B13:B12 |
| A15:A14 | B15:B14 |
| A17:A16 | B17:B16 |
| A19:A18 | B19:B18 |
| A21:A20 | B21:B20 |
| A23:A22 | B23:B22 |
| A25:A24 | B25:B24 |
| A27:A26 | B27:B26 |
| A29:A28 | B29:B28 |
| A31:A30 | B31:B30 |

**[0031]** Referring again to Figure 2, the eight functional units in processor 10's data paths can be divided into two groups of four; each functional unit in one data path is almost identical to the corresponding unit in the other data path. The functional units are described in Table 2.

Table 2.

| Functional Units and Operations Performed | |
|---|---|
| Functional Unit Operations | Fixed-Point |
| | |
| .L unit (.L1, .L2), 18a,b arithmetic and compare operations | 32/40-bit |
| 32-bit logical operations | |
| Leftmost 1 or 0 counting for 32 bits | |
| Normalization count for 32 and 40 bits | |
| shifts | Byte |
| packing/unpacking | Data |
| constant generation | 5-bit |
| Paired 16-bit arithmetic operations<br>8-bit arithmetic operations | Quad |
| Paired 16-bit min/max operations<br>8-bit min/max operations | Quad |
| .S unit (.S1, .S2) 16a,b operations | 32-bit arithmetic |
| 32/40-bit shifts and 32-bit bit-field operations | |
| 32-bit logical operations | |
| Branches | |
| Constant generation | |
| Register transfers to/from control register file (.S2 only) | |
| shifts | Byte |
| packing/unpacking | Data |
| Paired 16-bit compare operations<br>8-bit compare operations | Quad |
| Paired 16-bit shift operations<br>Paired 16-bit saturated arithmetic operations<br>8-bit saturated arithmetic operations | Quad |
| .M unit (.M1, .M2) 14a,b operations | 16 x 16 multiply |
| 32 multiply operations | 16 x |
| expansion | Bit |
| interleaving/de-interleaving | Bit |
| 8 x 8 multiply operations | Quad |
| | |
| Paired 16 x 16 multiply operations | |

Table 2. (continued)

| Functional Units and Operations Performed | |
|---|---|
| Functional Unit Operations | Fixed-Point |
| Paired 16 x 16 multiply with add/subtract operations<br><br>8 x 8 multiply with add operations | Quad |
| Variable shift operations<br><br>Rotation<br><br>Galois Field Multiply | |
| | |
| .D unit (.D1, .D2) 12a,b subtract, linear and circular address calculation | 32-bit add, |
| Loads and stores with 5-bit constant offset | |
| Loads and stores with 15-bit constant offset (.D2 only) | |
| Load and store double words with 5-bit constant | |
| Load and store non-aligned words and double words | |
| bit constant generation | 5- |
| bit logical operations | 32- |

**[0032]** Most data lines in the CPU support 32-bit operands, and some support long (40-bit) and double word (64-bit) operands. Each functional unit has its own 32-bit write port into a general-purpose register file 20a, 20b (Refer to Figure 2). All units ending in 1 (for example, .L1) write to register file A 20a and all units ending in 2 write to register file B 20b. Each functional unit has two 32-bit read ports for source operands src1 and src2. Four units (.L1, .L2, .S1, and .S2) have an extra 8-bit-wide port (long-dst) for 40-bit long writes, as well as an 8-bit input (long-src) for 40-bit long reads. Because each unit has its own 32-bit write port dst, when performing 32 bit operations all eight units can be used in parallel every cycle. Since each multiplier can return up to a 64-bit result, two write ports (dst1 and dst2) are provided from the multipliers to the register file.

Memory, Load and Store Paths

**[0033]** Processor 10 supports double word loads and stores. There are four 32-bit paths for loading data from memory to the register file. For side A, LD1a is the load path for the 32 LSBs (least significant bits); LD1b is the load path for the 32 MSBs (most significant bits). For side B, LD2a is the load path for the 32 LSBs; LD2b is the load path for the 32 MSBs. There are also four 32-bit paths, for storing register values to memory from each register file. ST1a is the write path for the 32 LSBs on side A; ST1b is the write path for the 32 MSBs for side A. For side B, ST2a is the write path for the 32 LSBs; ST2b is the write path for the 32 MSBs.

**[0034]** The ports for long and double word operands are shared between the S and L functional units. This places a constraint on which long or double word operations can be scheduled on a datapath in the same execute packet.

Data Address Paths

**[0035]** Bus 40a has an address bus DA1 which is driven by mux 200a. This allows an address generated by either load/store unit D1 or D2 to provide a memory address for loads or stores for register file 20a. Data Bus LD1a,b loads data from an address in memory 22 specified by address bus DA1 to a register in register file 20a. Likewise, data bus STla,b stores data from register file 20a to memory 22. Load/store unit D1 performs the following operations: 32-bit add, subtract, linear and circular address calculations. Load/store unit D2 operates similarly to unit Dl, with the assistance of mux 200b for selecting an address.

**[0036]** The DA1 and DA2 address resources and their associated data paths are connected to target memory ports on memory 22 specified as T1 and T2 respectively. T1 connects to the DA1 address path and the LD1a, LD1b, ST1a

and ST1b data paths. Similarly, T2 connects to the DA2 address path and the LD2a, LD2b, ST2a and ST2b data paths. The T1 and T2 designations appear in functional unit fields for load and store instructions.

**[0037]** For example, the following load instruction uses the .D1 unit to generate the address but is using the LD2a path resource with DA2 address bus connected to target port T2 to place the data in the B register file:

LDW .D1T2 *A0[3], B1.

The use of the DA2 address resource is indicated with the T2 designation.

Instruction Syntax

**[0038]** An instruction syntax is used to describe each instruction. An opcode map breaks down the various bit fields that make up each instruction. There are certain instructions that can be executed on more than one functional unit. The syntax specifies the functional unit and various resources used by an instruction, and typically has a form as follows: operation, .unit, src, dst. src and dst indicate source and destination operands respectively. The .unit dictates which functional unit the instruction is mapped to (.L1, .L2, .S1, .S2, .M1, .M2, .D1, or .D2). Several instructions have three opcode operand fields: src1, src2, and dst.

**[0039]** Figure 3A illustrates an opcode map for the load/store instructions which are executed in a .D unit of the DSP. Figure 3B illustrates an opcode map for non-aligned double word load/store instructions, and Figure 3C illustrates an opcode map for Boolean instructions executed by the .D units. Table 3 lists the opcodes for various load store (LD/ST) instructions performed by the CPU of the present embodiment. Opcode field 510 and R-field 512 define the operation of the LD/ST instructions. An aspect of the present invention is that processor 10 performs non-aligned load and store instructions by using resources of one D unit and both target ports T1 and T2, as will be described in more detail below. Advantageously, the second D unit is available to execute a Boolean or arithmetic instruction in parallel with the execution of a non-aligned load/store instruction.

**[0040]** The dst field of the LD/STNDW instruction selects a register pair, a consecutive even-numbered and odd-numbered register pair from the same register file. The instruction can be used to load a pair of 32-bit integers. The least significant 32 bits are loaded into the even-numbered register and the most significant 32 bits are loaded into the next register (which is always an odd-numbered register).

Table 3.

| Load/Store Instruction Opcodes | | | | |
|---|---|---|---|---|
| R-Opcode extension | LD/ST Op | Instruc tion | Size | Alignment |
| 0 | 000 | LDHU | Half word unsigned | Half word |
| 0 | 001 | LDBU | Byte unsigned | Byte |
| 0 | 010 | LDB | Byte | Byte |
| 0 | 011 | STB | Byte | Byte |
| 0 | 100 | LDH | Half word | Half |
| 0 | 101 | STH | Half word | Half word |
| 0 | 110 | LDW | Word | Word |
| 0 | 111 | STW | Word | Word |
| 1 | 010 | LDNDW | Double word | byte( non-aligned) |
| 1 | 011 | LDNW | Word | Byte (non-aligned) |
| 1 | 100 | STDW | Double word | Double word |
| 1 | 101 | STNW | Word | Byte (non-aligned) |
| 1 | 110 | LDDW | Double word | Double word |
| 1 | 111 | STNDW | Double word | Byte (non-aligned) |

Addressing Modes

**[0041]** The addressing modes are linear, circular using block size field BK0, and circular using block size field BK1. Eight registers can perform circular addressing. A4-A7 are used by the .D1 unit and B4-B7 are used by the .D2 unit. No other units can perform circular addressing modes. For each of these registers, an addressing mode register (AMR)

contained in control register file 102 specifies the addressing mode. The block size fields are also in the AMR.

**[0042]** Referring again to Figure 3A and 3B, linear mode addressing simply shifts the offsetR/cst operand 516 to the left by 3, 2, 1, or 0 for double word, word, half-word, or byte access respectively and then performs an add or subtract to baseR 514, depending on the address mode specified. For the pre-increment, pre-decrement, positive offset, and negative offset address generation options, the result of the calculation is the address to be accessed in memory. For post-increment or post-decrement addressing, the value of baseR before the addition or subtraction is the address to be accessed from memory. Address modes are specified by mode field 500 and listed in Table 4. The increment/decrement mode controls whether the updated address is written back to the register file. Otherwise, it is rather similar to offset mode. The pre-increment and offset modes differ only in whether the result is written back to "base". The post-increment mode is similar to pre-increment (e.g. the new address is written to "base"), but differs in that the old value of "base" is used as the address for the access. The same applies for negative offset vs. decrement mode.

Table 4 -

| Address Generator Options | | |
|---|---|---|
| Mode Field | Syntax | Modification Performed |
| 0 1 0 1 | *+R[ offsetR] | Positive offset; addr = base + offset * scale |
| 0 1 0 0 | *-R[ offsetR] | Negative offset; addr = base - offset * scale |
| 1 1 0 1 | *++R[ offsetR] | Preincrement; addr = base + offset * scale; base = addr |
| 1 1 0 0 | *--R[ offsetR] | Predecrement; addr = base - offset * scale; base = addr |
| 1 1 1 1 | *R++[ offsetR] | Postincrement; addr = base; base = base + offset * scale |
| 1 1 1 0 | *R--[ offsetR] | Postdecrement; addr = base; base = base - offset * scale |
| 0 0 0 1 | *+R[ ucst5] | Positive offset; addr = base + offset * scale |
| 0 0 0 0 | *-R[ ucst5] | Negative offset; addr = base - offset * scale |
| 1 0 0 1 | *++R[ ucst5] | Preincrement; addr = base + offset * scale; base = addr |
| 1 0 0 0 | *- -R[ ucst5] | Predecrement; addr = base - offset * scale; base = addr |
| 1 0 1 1 | *R++[ ucst5] | Postincrement; addr = base; base = base + offset * scale |
| 1 0 1 0 | *R- -[ ucst5] | Postdecrement; addr = base; base = base - offset * scale |

**[0043]** Figure 4 illustrates the addressing mode register, (AMR), which is included in control register file 102, is accessible via a "move between control file and the register file" (MVC) instruction. Eight registers (A4-A7, B4-B7) can perform circular addressing. For each of these registers, the AMR specifies the addressing mode. A 2-bit field for each register is used to select the address modification mode: linear (the default) or circular mode. With circular addressing, the field also specifies which BK (block size) field to use for a circular buffer. In this embodiment, the buffer must be aligned on a byte boundary equal to the block size. The mode select field encoding is shown in Table 5.

Table 5.

| Addressing Mode Field Encoding | |
|---|---|
| Mode | Description |
| 00 | Linear modification (default at reset) |
| 01 | Circular addressing using the BK0 field |
| 10 | Circular addressing using the BK1 field |
| 11 | Reserved |

**[0044]** The block size fields, BK0 and BK1, specify block sizes for circular addressing. The five bits in BK0 and BK1 specify the width. The formula for calculating the block size width is:

$$\text{Block size (in bytes)} = 2^{(N+1)}$$

where N is the value in BK1 or BK0

**[0045]** Table 6 shows block size calculations for all 32 possibilities.

Table 6.

| Block Size Calculations | | | |
|---|---|---|---|
| N | Block Size | N | Block Size |
| 00000 | 2 | 10000 | 131,072 |
| 00001 | 4 | 10001 | 262,144 |
| 00010 | 8 | 10010 | 524,288 |
| 00011 | 16 | 10011 | 1,048,576 |
| 00100 | 32 | 10100 | 2,097,152 |
| 00101 | 64 | 10101 | 4,194,304 |
| 00110 | 128 | 10110 | 8,388,608 |
| 00111 | 256 | 10111 | 16,777,216 |
| 01000 | 512 | 11000 | 33,554,432 |
| 01001 | 1?024 | 11001 | 67,108,864 |
| 01010 | 2?048 | 11010 | 134,217,728 |
| 01011 | 4?096 | 11011 | 268,435,456 |
| 01100 | 8?192 | 11100 | 536,870,912 |
| 01101 | 16?384 | 11101 | 1,073,741,82 4 |
| 01110 | 32?768 | 11110 | 2,147,483,64 8 |
| 01111 | 65?536 | 11111 | 4,294,967,29 6 |
| Note: when N is 11111, the behavior is identical to linear addressing | | | |

**[0046]** Circular mode addressing uses the BK0 and BK1 fields in the AMR to specify block sizes for circular addressing. Circular mode addressing operates as follows with LD/ST Instructions: after shifting offsetR/cst to the left by 3, 2, 1, or 0 for LDDW, LDW, LDH, or LDB respectively, and is then added to or subtracted from baseR to produce the final address. This add or subtract is performed by only allowing bits N through 0 of the result to be updated, leaving bits 31 through N+1 unchanged after address arithmetic. The resulting address is bounded to 2^(N+1) range, regardless of the size of the offsetR/cst.

**[0047]** The circular buffer size in the AMR is not scaled; for example: a size of 8 is 8 bytes, not 8 × size of (type). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. Table 7 shows an example LDW instructions performed with register A4 in circular mode, with BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value put in the AMR for this example is 00040001h. In this example, an offset of "9" is specified. 9h (hexadecimal) words is 24h bytes. 24h bytes is 4 bytes beyond the 32-byte (20h) boundary 100h-11Fh; thus, it is wrapped around to (124h - 20h = 104h).

Table 7. LDW in Circular Mode

| LDW .D1 *++A4[9],A1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Before LDW | | | 1 cycle after LDW | | | 5 cycles after LDW | |
| | A4 | 0000 0100h | | A4 | 0000 0104h | | A4 | 0000 0104h |
| | | | | | | | | |
| | A1 | XXXX XXXXh | | A1 | XXXX XXXXh | | A1 | 1234 5678h |

Table 7. LDW in Circular Mode (continued)

| LDW .D1 *++A4[9],A1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Mem | 1234 | | mem 104h | 1234 5678h | | mem 104h | 1234 | |
| | Before LDW | | 1 cycle after LDW | | | 5 cycles after LDW | | |
| 104h | 5678h | | | | | | 5678h | |

Non-Aligned Memory Access Considerations

**[0048]** Circular addressing may be used with non-aligned accesses. When circular addressing is enabled, address updates and memory accesses occur in the same manner as for the equivalent sequence of byte accesses. The only restriction is that the circular buffer size be at least as large as the data size being accessed. Non-aligned access to circular buffers that are smaller than the data being read produce undefined results.

**[0049]** Non-aligned accesses to a circular buffer apply the circular addressing calculation to logically adjacent memory addresses. The result is that non-aligned accesses near the boundary of a circular buffer will correctly read data from both ends of the circular buffer, thus seamlessly causing the circular buffer to "wrap around" at the edges.

**[0050]** Figures 5A, 5B and 5C illustrate aspects of non-aligned address formation and non-aligned data extraction from a circular buffer region, according to an aspect of the present invention. Consider, for example, a circular buffer 500 that has a size of 16 bytes illustrated in Figure 5A. A circular buffer of this size is specified by setting either BK0 or BK1 to "00011." For example with register A4 in circular mode and BK0 = 3, the buffer size is 16 bytes, 8 half words, or 4 words. The value put in the AMR for this example is 00030001h. The buffer starts at address 0x0020 (502) and ends at 0x002F (504). The register A4 is initialized to the address 0x0028, for example; however, the buffer could be located at other places in the memory by setting more significant address bits in register A4. Below the buffer at address 0x1F (506) and above the buffer at address 0x30 (508) data can be stored that is not relevant to the buffer.

**[0051]** The effect of circular buffering is to make it so that memory accesses and address updates in the 0x20 - 0x2F range stay completely inside this range. Effectively, the memory map behaves as illustrated in Figure 5B. Executing a LDW instruction with an offset of 1 in post increment mode will provide an address of 0x0028 (511) and access word 510, for example. Executing the instruction a second time will provide an address of 0x002C (513) and access word 512 at the end of the circular buffer. Executing the instruction a third time will provide an address of 0x0020 (502a) and access word 514. Note that word 514 actually corresponds to the other end of the circular buffer, but was accessed by incrementing the address provided by the LDW instruction.

**[0052]** Figure 5C illustrates the operation of an access into the circular buffer using a non-aligned load/store instruction. In this example, A4 is initialized to the address 0x002A and a non-aligned double word load instruction (LDNDW) with a non-scaled offset of "1" in post increment mode. As discussed above, two addresses will be sent to the two ports on the memory. An address of 0x002A (534) will be sent on the first port, which results in accessing an aligned double word DW1 from memory, aligned on address 0x0028 (530). A second address will be sent to the memory system that is incremented by the line size of the instruction. Since in this example the instruction is a double word instruction, the line size is two words, or eight bytes. Thus the second address in incremented by eight bytes to be 0x0032. However, according to an aspect of the present invention, this address is bounded to 0x0022 (536) by circular addressing circuitry to remain within the bounds of circular buffer region 500. The memory system then accesses a second aligned double word DW2, aligned at address 0x0020 (532). Extraction circuitry then extracts non-aligned double word NADW1 from the two logically adjacent double words DW1 and DW2, even though they are actually physically from different ends of the circular buffer.

**[0053]** Still referring to Figure 5C, executing the LDNDW instruction again results in sending a first address of 0x002B (538) incremented by a non-scaled offset of "1" and a second address of 0x0023 (540) incremented by the line size and bounded to remain within the circular buffer region to the memory system. The memory system will access the same two aligned double words DW1, DW2. However, the extraction circuitry now extracts non-aligned double word NADW2 in response in response to incremented address 538.

**[0054]** As another example, Table 8 shows an LDNW performed with register A4 in circular mode and BK0 = 3, so the buffer size is 16 bytes, 8 half words, or 4 words. The value put in the AMR for this example is 00030001h. The buffer starts at address 0x0020 and ends at 0x002F. The register A4 is initialized to the address 0x002A. In this example, on offset of "2" is specified. 2h words is 8h bytes. 8h bytes is 3 bytes beyond the 16 byte (10h) boundary starting at address 002Ah; thus, it is wrapped around to 0022h (002Ah + 8h = 0022h). In this example, the two address sent to the memory subsystem are contiguous; the first address is 0x0022 and the second address is incremented by a line size of 4h, to become 0x0026.

Table 8.

| LDNW in Circular Mode | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| LDNW .D1 *++A4[2],A1 | | | | | | | | |
| | Before LDW | | | 1 cycle after LDW | | | 5 cycles after LDW | |
| | A4 | 0000 002Ah | | A4 | 0000 0022h | | A4 | 0000 00022h |
| | | | | | | | | |
| | A1 | XXXX XXXXh | | A1 | XXXX XXXXh | | A1 | 5678 9ABCh |
| Mem 0022h | | 5678 9ABCh | | mem 0022h | 5678 9ABCh | | mem 0022h | 5678 9ABCh |

**[0055]** Figure 6 illustrates the basic format of a fetch packet of the DSP. In this embodiment, instructions are always fetched eight at a time. This constitutes a fetch packet. The execution grouping of the fetch packet is specified by the p-bit, bit zero, of each instruction. Fetch packets are 8-word aligned and can contain up to eight instructions. A p bit in each instruction controls the parallel execution of instructions. A set of instructions executing in parallel constitute an execute packet. An execute packet can contain up to eight instructions.

**[0056]** The p bit controls the parallel execution of instructions. The p bits are scanned from left to right (lower to higher address) . If the p bit of instruction i is 1, then instruction i +1 is to be executed in parallel with (in the same cycle as) instruction i. If the p-bit of instruction i is 0, then instruction i + 1 is executed in the cycle after instruction i. All instructions executing in parallel constitute an execute packet. An execute packet can contain up to eight instructions. All instructions in an execute packet must use a unique functional unit.

**[0057]** There are three types of p-bit patterns for fetch packets which result in the following execution sequences for the eight instructions: fully serial; fully parallel, or partially parallel. As discussed above, in this embodiment of the invention, a non-aligned fetch or store instruction uses only one .D unit, so that the other .D unit is available for parallel execution of an instruction that does not access a load/store port on memory 22. The processor can access words and double words at any byte boundary using non-aligned loads and stores. As a result, word and double word data does not always need alignment to 32-bit or 64-bit boundaries.

**[0058]** As an example of parallel execution of load/store instructions, the following execute packet is invalid in the present embodiment because there are two memory operations and one of them is non-aligned:
LDNW .D2T2 *B2[B12],B13; II LDB .D1T1 *A2,A14; However, the following execute packet is valid in the present embodiment because there is a non-memory Boolean/arithmetic instruction being executed on .D1 in parallel with a non-aligned load instruction on .D2:
LDNW .D2T2 *B2[B12], A13; II ADD .D1x A12, B13, A14;

Pipeline Operation

**[0059]** The instruction execution pipeline of DSP 1 has several key features which improve performance, decrease cost, and simplify programming, including: increased pipelining eliminates traditional architectural bottlenecks in program fetch, data access, and multiply operations; control of the pipeline is simplified by eliminating pipeline interlocks; the pipeline can dispatch eight parallel instructions every cycle; parallel instructions proceed simultaneously through the same pipeline phases; sequential instructions proceed with the same relative pipeline phase difference; and load and store addresses appear on the CPU boundary during the same pipeline phase, eliminating read-after-write memory conflicts.

**[0060]** A multi-stage memory pipeline is present for both data accesses in memory 22 and program fetches in memory 23. This allows use of high-speed synchronous memories both on-chip and off-chip, and allows infinitely nestable zero-overhead looping with branches in parallel with other instructions.

**[0061]** There are no internal interlocks in the execution cycles of the pipeline, so a new execute packet enters execution every CPU cycle. Therefore, the number of CPU cycles for a particular algorithm with particular input data is fixed. If during program execution, there are no memory stalls, the number of CPU cycles equals the number of clock cycles for a program to execute.

**[0062]** Performance can be inhibited by stalls from the memory system, stalls for cross path dependencies, or interrupts. The reasons for memory stalls are determined by the memory architecture. Cross path stalls are described in

detail in U.S. Patent Application No.09/702,456, to Steiss, et al. To fully understand how to optimize a program for speed, the sequence of program fetch, data store, and data load requests the program makes, and how they might stall the CPU should be understood.

**[0063]** The pipeline operation, from a functional point of view, is based on CPU cycles. A CPU cycle is the period during which a particular execute packet is in a particular pipeline stage. CPU cycle boundaries always occur at clock cycle boundaries; however, memory stalls can cause CPU cycles to extend over multiple clock cycles. To understand the machine state at CPU cycle boundaries, one must be concerned only with the execution phases (E1-E5) of the pipeline. The phases of the pipeline are described in Table 9.

Table 9.

| Pipeline Phase Description | | | | |
|---|---|---|---|---|
| Pipeline | Pipeline Phase | Symbol | During This Phase | Instruction Types Completed |
| Program Fetch | Program Address Generate | PG | Address of the fetch packet is determined. | |
| | Program Address Send | PS | Address of fetch packet is sent to memory. | |
| | Program Wait | PW | Program memory access is performed. | |
| | Program Data Receive | PR | Fetch packet is expected at CPU boundary. | |
| Program Decode | Dispatch | DP | Next execute packet in fetch packet determined and sent to the appropriate functional units to be decoded. | |
| | Decode | DC | Instructions are decoded at functional units. | |
| Execute | Execute 1 | E1 | For all instruction types, conditions for instructions are evaluated and operands read.<br>Load and store instructions: address generation is computed and address modifications written to register file†<br>Branch instructions: affects branch fetch packet in PG phase†<br>Single-cycle instructions: results are written to a register file† | Single-cycle |

†This assumes that the conditions for the instructions are evaluated as true. If the condition is evaluated as false, the instruction will not write any results or have any pipeline operation after E1.

Table 9. (continued)

| Pipeline Phase Description | | | | |
|---|---|---|---|---|
| Pipeline | Pipeline Phase | Symbol | During This Phase | Instruction Types Completed |
| | Execute 2 | E2 | Load instructions: address is sent to memory† Store instructions and STP: address and data are sent to memory† Single-cycle instructions that saturate results set the SAT bit in the Control Status Register (CSR) if saturation occurs. † Multiply instructions: results are written to a register file† | Stores STP Multiplies |
| | Execute 3 | E3 | Data memory accesses are performed. Any multiply instruction that saturates results sets the SAT bit in the Control Status Register (CSR) if saturation occurs. † | |
| | Execute 4 | E4 | Load instructions: data is brought to CPU boundary† | |
| | Execute 5 | E5 | Load instructions: data is loaded into register† | Loads |

†This assumes that the conditions for the instructions are evaluated as true. If the condition is evaluated as false, the instruction will not write any results or have any pipeline operation after E1.

**[0064]** The pipeline operation of the instructions can be categorized into seven types shown in Table 10. The delay slots for each instruction type are listed in the second column.

Table 10.

| Delay Slot Summary | | |
|---|---|---|
| Instruction Type | Delay Slots | Execute Stages Used |
| Branch (The cycle when the target enters E1) | 5 | E1-branch target E1 |
| Load (LD) (Incoming Data) | 4 | E1 - E5 |
| Load (LD) (Address Modification) | 0 | E1 |
| Multiply | 1 | E1 - E2 |
| Single-cycle | 0 | E1 |
| Store | 0 | E1 |
| NOP (no execution pipeline operation) | - | - |
| STP (no CPU internal results written) | - | - |

**[0065]** The execution of instructions can be defined in terms of delay slots (Table 10). A delay slot is a CPU cycle

that occurs after the first execution phase (E1) of an instruction in which results from the instruction are not available. For example, a multiply instruction has 1 delay slot, this means that there is 1 CPU cycle before another instruction can use the results from the multiply instruction.

**[0066]** Single cycle instructions execute during the EI phase of the pipeline. The operand is read, operation is performed and the results are written to a register all during E1. These instructions have no delay slots.

**[0067]** Load instructions have two results: data loaded from memory and address pointer modification.

**[0068]** Data loads complete their operations during the E5 phase of the pipeline. In the E1 phase, the address of the data is computed. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory read is performed. In the E4 stage, the data is received at the CPU core boundary. Finally, in the E5 phase, the data is loaded into a register. Because data is not written to the register until E5, these instructions have 4 delay slots. Because pointer results are written to the register in E1, there are no delay slots associated with the address modification.

**[0069]** Store instructions complete their operations during the E3 phase of the pipeline. In the E1 phase, the address of the data is computed. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory write is performed. The address modification is performed in the E1 stage of the pipeline. Even though stores finish their execution in the E3 phase of the pipeline, they have no delay slots and follow the following rules ( i = cycle):

1) When a load is executed before a store, the old value is loaded and the new value is stored.
2) When a store is executed before a load, the new value is stored and the new value is loaded.
3) When the instructions are in are in parallel, the old value is loaded and the new value is stored.

**[0070]** As discussed earlier, in this embodiment of the present invention non-aligned load and store instructions are performed by using resources of one D unit and both target ports T1 and T2, as will be described in more detail below. Advantageously, the second D unit is available to execute a Boolean or arithmetic instruction in parallel with the execution of a non-aligned load/store instruction. Aspects of non-aligned memory accesses will now be described in more detail.

**[0071]** Figure 7 is a memory map of a portion of the memory space of the DSP 1 and illustrates various aligned and non-aligned memory accesses. This portion of memory can be at any address YYYYYNNXh, but only the portion of the address represented by NNXh will be referred to herein, for convenience. Furthermore, the addresses used in the following discussion are only for example and are not intended to limit the invention in any manner.

**[0072]** DSP 1 can access both target ports T1, T2 of data memory 22 by executing two aligned load or store instructions in parallel, as discussed above. For example, a double word 700 at address 700h and a double word 708 at address 708h can be accessed by two load double word (LDDW) instructions executed in parallel using .D1 and .D2 and target ports T1 and T2. Likewise, word 780 and half word 786 can be accessed by executing a load word (LDW) instruction and a load half word (LDH) instruction in parallel using .D1 and .D2 and target ports T1 and T2.

**[0073]** Advantageously, this embodiment of the present invention utilizes the two target ports and two address buses DA1, DA2 to perform a non-aligned access. For example, double word 721 at address 721h is non-aligned by one byte. Double word 74Da-74Db at address 74Dh is located in two different rows of the memory. Single word 7B7 located at address 7B7h is non-aligned by three bytes. Advantageously, each non-aligned access is performed in the same amount of time as each aligned access, unless the data word is not present in memory 22 and must be retrieved from secondary memory storage, such as off-chip memory 70 of Figure 1.

**[0074]** Uniform access time is important for software programs that operate in real time, such as are commonly executed on DSPs. The problem for real time comes when a loop walks a data structure by a stride related to the cache/SRAM line size. If the structure starts at an offset such that the unaligned access doesn't require access outside of the single line, the loop runs quickly since every access runs without the stall. If the starting offset is such that the nonaligned load crosses the line boundary, there is a stall on every access. The same loop might run twice as long this time. If a real-time system is designed for the longer loop time, then twice as much performance is being sacrificed most of the time.

**[0075]** Figure 8 is a block diagram illustrating D-unit address buses of DSP 1 in more detail and illustrating two target ports T1, T2 of DSP memory 22. An aspect of the embodiment of the present invention is that load/store unit .D1 can generate an address for a non-aligned access and provide it on address bus DA1 via address signals 800 and multiplexer 200a, and simultaneously generate a contiguous address that is greater by the data size and provide it on address bus DA2 via address signals 801 and multiplexer 200b, as depicted in Table 11. In this embodiment of the invention, load/store unit .D2 can also generate an address for a non-aligned access and simultaneously generate a contiguous address incremented by the data size and provide them to address buses DA1 and DA2 via address signal lines 810 and 811 and multiplexors 200a and 200b, respectively. However, in an alternative embodiment, only one load/store unit may be so equipped. In yet another embodiment, there may be more than two D units so equipped, for example.

**[0076]** In this embodiment of the invention, DSP 1 supports non-aligned memory loads and stores for words and

doublewords. Only one non-aligned access can be performed in a single cycle because both target ports T1, T2 are used to load/store part of the data. From the memory designer's perspective, the two memory operations due to a non-aligned access are indistinguishable from two memory operations resulting from two instructions executed in parallel and in both cases the same memory ordering properties apply. The DSP simply requests an aligned access to each target port T1, T2 and byte strobes accompany data that must be written. Table 11 shows the accesses that are performed as a result of non-aligned accesses. Alternative embodiments of the present invention may support other data sizes for non-aligned access. An alternative embodiment of the present invention may provide the addresses in another form, such as a byte address without being truncated to the nearest word address, for example. Advantageously, memory 22 bank conflicts do not occur during non-aligned access.

Table 11.

| Non-Aligned Memory Access | | | |
|---|---|---|---|
| Request Size | Non-Aligned Byte Offset | DA2 Byte Address | DA1 Byte Address |
| Word | 0x1 to 0x3 | 0x4 | 0x0 |
| Word | 0x5 to 0x7 | 0x8 | 0x4 |
| Word | 0x9 to 0xB | 0xC | 0x8 |
| Word | 0xD to 0xF | 0x10 | 0xC |
| Word | 0x11 to 0x13 | 0x14 | 0x10 |
| Word | 0x15 to 0x17 | 0x18 | 0x14 |
| Word | 0x19 to 0x1B | 0x1C | 0x18 |
| Word | 0x1D to 0x1F | 0x20 | 0x1C |
| Doubleword | 0x1 to 0x7 | 0x8 | 0x0 |
| Doubleword | 0x9 to 0xF | 0x10 | 0x8 |
| Doubleword | 0x11 to 0x17 | 0x18 | 0x10 |
| Doubleword | 0x19 to 0x1F | 0x20 | 0x18 |

**[0077]** Figure 9 is a block diagram of the memory of Figure 8 illustrating address decoding of the two target ports T1, T2 and byte selection circuitry to extract a non-aligned data item according to an aspect of the present invention. Byte selection circuitry 910 selects data from a set of memory banks 940-947 and provides the selected data to load data signals 901 and to load data signals 902 that are connected respectively to load data buses LD1a,b and LD2a, b. In this embodiment of the present invention, there are eight memory banks 940-947 that each store sixteen bits of data, so that two sets of 64 bit data can be selected and provided on load data signals 901, 902. Address ports 921 and 922 each receive an address from address buses DA1 and DA2, respectively and provide a portion of the address to separate inputs on address multiplexers 950-957 that provide addresses to the memory banks. Decode circuitry 930 decodes a portion of the address MSBs to determine that a memory request is intended for memory 22. Decode signals 932 are formed by decoder 930 and sent to address multiplexors 950-957 to select which address is provided to each memory bank.

**[0078]** Decode circuitry 930 also receives a set of control signals 931 from instruction decode circuitry 10c of DSP 1 to identify if a non-aligned access is being processed by memory 22. In response to control signals 931 and four LSB address bits from each address bus DA1, DA2, decode circuitry 930 forms byte selection signals 933 that are sent to byte selection circuitry 910. When one or two aligned load requests are being executed, byte selection circuitry places the requested byte, half word, word or double word on the appropriate set of load data signals 901, 902 in a right aligned manner in response to byte selection signals 933.

**[0079]** When a non-aligned load request is being executed, byte selection circuitry 910 places the selected word or double word on the appropriate set of load data signals 901 or 902 in response to byte selection signals 933. For example, referring back to Figure 7, for non-aligned double word access 74D, memory banks 946 and 947 are accessed at aligned address 748h provided on address bus DA1 and three bytes are selected corresponding to byte addresses 74Dh-74Fh. Memory banks 940, 941, and 942 are accessed at contiguous aligned address 750h provided on address bus DA2 and five bytes are selected corresponding to byte addresses 750h-754h. Note that the address provided on DA2 is a value of 8h greater than the aligned address on DA1, corresponding to the eight byte size of the requested non-aligned data item. These eight bytes are then right aligned and provided on load data signals 901 if register file A

20a is the specified destination of the transfer or on load signals 902 if register file 20b is the specified destination of the transfer. In this embodiment, the load data bus LDx that is not associated with the specified destination register file remains free so that an associated .S unit can use the shared register file write port.

**[0080]** Figure 10 is a block diagram illustrating load byte selection circuitry 910, also referred to as extraction circuitry, of Figure 9 in more detail. For simplicity, only byte select multiplexors 1000-1007 connected to load data byte lanes 901(0)-901(7) are shown for simplicity. Another similar set of multiplexors is connected to load data signals 902. Selected ones of byte selection signals 933 are connected to each multiplexor to select the appropriate one of sixteen bytes provided by the memory bank array.

**[0081]** Figure 11 is a block diagram illustrating the store byte selection circuitry of the memory system Figure 8 in more detail. For purposes of this document, the store byte selection circuitry is also referred to as insertion circuitry for storing a non-aligned data item into the memory subsystem. Pipe 1 store data signals 1121 provide store data from store data buses ST1a,b to byte selection multiplexors 1100-1115. Likewise, Pipe 2 store data signals 1122 provide store data from store data buses ST2a,b to byte selection multiplexors 1100-1115. Control signals (not shown) provided to each byte multiplexor from decode circuitry 930 selects the appropriate one of sixteen bytes and presents each selected byte to the respective memory bank 940-947. Write signals byte0-byte15 are asserted as appropriate to cause a selected byte to be written into the respective memory bank.

**[0082]** In this embodiment of the present invention, the load byte selection circuitry and the store byte selection circuitry is required to support the various aligned accesses available via each of the target ports T1, T2. Advantageously, a single non-aligned access can be supported with only minor changes to the byte selection circuitry. Advantageously, all of the memory address decoding circuitry and memory banks do not need any modification and execute a non-aligned access simply as two aligned accesses in response to the two addresses provided on address buses DA1 and DA2.

**[0083]** Figure 12A is a block diagram of a load/store .D unit, which executes the load/store instructions and performs address calculations. The .D unit receives a base address via first source input src1. An offset value can be selected from either a second source input src2 or from a field in the instruction opcode, indicated at 1200. An address is provided on address output 1202 that is in turn connected to at least one of address multiplexors 200a,b. Additionally, an augmented address is provided on address output 1204 for non-aligned accesses. The augmented address is incremented by a byte address value of either four or eight as selected by multiplexer 1210 in response to the line size of the instruction being executed: four is selected for a word instruction and eight is selected for a double word instruction. Adder 1212 increments an address on signal lines 1213 by the amount selected by multiplexer 1210 to form the augmented address that is provided on signal lines 1214. This contiguous address is provided on address output 1204 for a non-aligned access and is connected to the other address multiplexor 200a,b, as discussed previously. A calculated address value is also provided to the output dst to update a selected base address register value in the register file when an increment or decrement address mode is selected. According to an aspect of the present invention, the address on signal lines 1213 and the augmented address on signal lines 1214 are passed through circular buffer circuitry 1230 prior to being output on 1202, 1204 so that they can be bounded to remain within a circular buffer region.

**[0084]** In this embodiment, Load and Store instructions operate on data sizes from 8 bits to 64 bits. Addressing modes supported by the .D unit are basic addressing, offset addressing, scaled addressing, auto-increment/auto-decrement, long-immediate addressing, and circular addressing, as defined by mode field 500. In basic addressing mode, the content of a selected base register is used as a memory address. In offset addressing mode, the memory address is determined by two values, a base value and an offset that is either added or subtracted from the base. Referring again to Figure 3A and Figure 3B, the base value always comes from a base register specified by a field 514 "base R" that is any of the registers in the associated register file 20a or 20b, whereas the offset value may come from either a register specified by an "offset R" field 516 or a 5-bit unsigned constant UCST5 contained in field 516 of the instruction via signals 1200. Certain load/store instructions have a long immediate address mode that uses a 15-bit unsigned constant contained in the instruction (not shown in Figure 3). A selected offset is provided on signal lines 1218 to shifter 1220. Scaled addressing mode functions the same as offset addressing mode, except that the offset is interpreted as an index into a table of bytes, half-words, words or double-words, as indicated by the data size of the load or store operation, and the offset is shifted accordingly by shifter 1220 in response to control signals 1226 which are derived by decoding opcode field 510, 512 of the LD/ST instructions.

**[0085]** In this embodiment of the present invention, an SC bit 520 in load/store non-aligned double word (LDNDW/STNDW) instruction controls shifter 1220 so that an offset can be used directly, referred to as unscaled, or shifted by an amount corresponding to the type of instruction, referred to as scaled. Scaled/unscaled control signal 1224 is derived by decoding the SC field 520 of LDNDW/STNDW instructions. If SC field 520 is a logical 0, then the offset is not scaled and signal 1224 is deasserted. If SC field 520 is a logical 1, then the offset is scaled and signal 1224 is asserted. In this embodiment, for instructions other than LDNDW/STNDW, signal 1224 is asserted so that scaling will be performed according to data size control signals 1226.

**[0086]** In auto-increment/decrement addressing mode, the base register is incremented/ decremented after the ex-

ecution of the load/store instruction by inc/dec unit 1222. There are two sub-modes, pre-increment/decrement, where the new value in the base register is used as the load/store address, and post-increment/decrement where the original value in the register is used as the load/store address. In long-immediate addressing mode, a 15-bit unsigned constant is added to a base register to determine the memory address. In circular addressing mode, the base register along with a block size define a region in memory. To access a memory location in that region, a new index value is generated from the original index modulo the block size in circular addressing unit 1230.

**[0087]** In this embodiment of the invention, a Boolean unit 1240 is provided and can be used for execution of logical instructions when the .D unit is not being used to generate an address.

**[0088]** Figure 12B is a more detailed block diagram of circular buffer circuitry 1230 of Figure 12A. As explained earlier, circular mode addressing operates as follows with LD/ST Instructions: after shifting offsetR/cst to the left by 3, 2, 1, or 0 for LDDW, LDW, LDH, or LDB respectively, it is then added to or subtracted from baseR to produce the final address. This add or subtract is performed by only allowing bits N through 0 of the result to be updated, leaving bits 31 through N+1 unchanged after address arithmetic. The resulting address is bounded to 2^(N+1) range, regardless of the size of the offsetR/cst. Bounding can be performed in a number of ways, such as by interrupting a carry bit at the appropriate place of adder 1222. However, in order to support non-aligned accesses, both the address and the augmented address must be bounded separately.

**[0089]** In the present embodiment, bounding circuitry 1250 bounds the address provided on signal lines 1213, while bounding circuitry 1260 bounds the augmented address provided on signal lines 1214. Mask generation circuit 1232 forms a right extended mask (R-mask) in response to a selected block size from the AMR register, as described earlier, and provides it on bus 1234. A right extended mask has a "1" in every bit position corresponding to an address bit within the bounds of the 2^(N+1) range, and a "0" in every more significant address bit beyond this range.

**[0090]** The R-mask is bit-wise ANDed with the address on bus 1213 in AND block 1252 to form a least significant portion of the address bounded within the 2^(N+1) range. An inverted R-mask is bit-wise ANDed with the original base address on bus 1216 in AND block 1254 to form a most significant portion of the address above the 2^ (N+1) range. The most significant address portion and the bounded least significant address portion are bit-wise combined in OR block 1256 to form the final address that is output on bus 1215. The augmented address on bus 1214 is likewise bounded using AND blocks 1262, 1264 and OR block 1266 and then output on bus 1217.

**[0091]** Advantageously, by having two bounding circuits 1250, 1260 both address are formed in a parallel manner so that a non-aligned access to a circular buffer region is performed in the same amount of time as an aligned access to a circular buffer region.

**[0092]** Figure 12C is a flow chart illustrating formation of scaled and non-scaled addresses for accessing a linear region or a circular buffer region with either aligned or non-aligned accesses, according to an aspect of the present invention. In step 600, a circular buffer region is setup in memory subsystem 22 by initializing the AMR register and an associated base register, as discussed above.

**[0093]** In step 602, an instruction is fetched for execution. In this embodiment of the present invention, instructions are fetched in fetch packets of eight instructions simultaneously during instruction execution pipeline phases P/G, PS, PW and PR. Other embodiments of the present invention may fetch instructions singly or doubly, for example, in a different number of phases.

**[0094]** In step 610, the instruction is decoded to form a plurality of fields. In this embodiment, decoding is performed in two phases of the instruction execution pipeline, but in other embodiments of the present invention decoding may be performed on one or three or more phases.

**[0095]** In step 620, a base-offset address for accessing a data item for the instruction is formed by combining in 627 a base address value and an offset value, such that the offset value is selectively scaled or not scaled. Step 627 may include post or pre-incrementing or decrementing, for example, as indicated by mode field 500. In 621 or 622, for a non-aligned double word load or store instruction (LD/STNDW) the offset value is scaled by shifting left three bits only if the SC field 520 has a value of 1. If SC field 520 has a value of 0, then the offset value is not scaled and is therefore treated as a byte offset. If the instruction is a load or store double, then the offset is scaled by left shifting three bits in step 623 to form a double word offset. If the instruction is a LD/ST word, then the offset is scaled by shifting left two bits in step 624 to form a word offset. If the instruction is a half word LD/ST instruction, then the offset is scaled by shifting left one bit in step 625 to form a half word offset. If the instruction is a byte LD/ST instruction, then the offset is scaled by shifting zero bits in step 626 to form a byte offset. In the present embodiment, the scaling amount is determined by opcode field 510, 512 that specifies the type of LD/ST instruction. In another embodiment, there may be a field to specify operand size, for example. In the present embodiment, step 620 is performed during the E1 pipeline phase.

**[0096]** In step 630, if the instruction fetched in step 602 is an aligned type instruction, then the base-offset address from step 627 is concatenated to stay within the boundary of the circular buffer region specified in step 602, if circular addressing is specified by the AMR for the base register selected by the instruction. In step 632, the resultant address is sent to the memory subsystem during pipeline phase E2. If circular addressing is not selected, then the base-offset

from step 627 is used to access memory during pipeline phase E2.

**[0097]** In step 640, if the instruction fetched in step 602 is a non-aligned type instruction, then a line size is added to the base-offset address from step 627 to form an augmented address. The line size is determined by the instruction type decoded in step 610. For a double word instruction type, the line size is eight bytes. For a word instruction type, the line size is four bytes.

**[0098]** In step 650, the base-offset address from step 627 is concatenated to stay within the boundary of the circular buffer region specified in step 602, if circular addressing is specified by the AMR for the base register selected by the instruction. In step 652, the resultant address is sent to the first port of the memory subsystem during pipeline phase E2. If circular addressing is not selected, then the base-offset from step 627 is used to access memory during pipeline phase E2. Likewise, in steps 651 and 653, the augmented is selectively bounded if circular addressing is selected and a second address is sent to the second port of the memory subsystem during pipeline phase E2.

**[0099]** During step 654, the requested non-aligned data item is extracted from the two aligned data items accessed in steps 652, 653.

**[0100]** An assembler which supports this embodiment of the invention defaults increments and decrements to 1 and offsets to 0 if an offset register or constant is not specified. Loads that do not modify to the baseR can use the assembler syntax *R. Square brackets, [ ], indicate that the ucst5 offset is left-shifted by 3 for double word loads. Parentheses, ( ), are be used to tell the assembler that the offset is a non-scaled offset. For example, LDNDW (.unit) *+baseR (14), dst represents an offset of 14 bytes and the assembler writes out the instruction with offsetC = 14 and sc = 0. Likewise, LDNDW (.unit) *+baseR [16] dst represents an offset of 16 double words, or 128 bytes, and the assembler writes out the instruction with offsetC = 16 and sc = 1.

**[0101]** In this embodiment, LD/STDW instructions do not include an SC field, but LDNDW and STNDW instruction do include an SC field. However, parentheses, ( ), are used to tell the assembler that the offset is a non-scaled, constant offset. The assembler right shifts the constant by 3 bits for double word stores before using it for the ucst5 field. After scaling by the STDW instruction, this results in the same constant offset as the assembler source if the least significant three bits are zeros. For example, STDW (.unit) src, *+baseR (16) represents an offset of 16 bytes (2 double words), and the assembler writes out the instruction with ucst5 = 2. STDW (.unit) src, *+baseR [16] represents an offset of 16 double words, or 128 bytes, and the assembler writes out the instruction with ucst5 = 16.

**[0102]** Referring again to step 620 of Figure 6, the SC bit (scale or not scaled) affects pre/post incrementing. If a pre or post increment/ decrement is specified, then the increment/decrement amount is controlled by the SC bit. In non-scaled mode, the increment/decrement corresponds to a number of bytes. In assembly code, this would be written as shown in Table 12, example 1 and 2. In both of these cases, regl ends up with the value "regl + reg2".

**[0103]** In scaled mode, the increment/decrement corresponds to a number of double-words. The assembly syntax for this is shown in Table 12, examples 3 and 4. In both of these cases, regl ends up with the value "regl + 8*reg2". That is, reg2 is "scaled" by the size of the access.

**[0104]** These comments also apply to the integer offset modes as well, as illustrated in Table 12, examples 5-8. Likewise, similar examples apply to the pre/post decrement instructions.

Table 12.

| Examples of Instructions With Various Pre/Post Increment, Scaled and Non-Scaled Addressing Modes | | |
|---|---|---|
| example | Instruction syntax | operation |
| 1 | LDNDW<br>*++reg1(reg2), reg3 | pre-increment, non-scaled |
| 2 | LDNDW<br>*reg1++(reg2), reg3 | post-increment, non-scaled |
| 3 | LDNDW<br>*++reg1[reg2], reg3 | pre-increment, scaled. |
| 4 | LDNDW<br>*reg1++[reg2], reg3 | post-increment, scaled. |
| 5 | LDNDW<br>*++reg1(cst5), reg2 | pre-increment, non-scaled |
| 6 | LDNDW<br>*reg1++(cst5), reg2 | post-increment, non-scaled |

Table 12. (continued)

| Examples of Instructions With Various Pre/Post Increment, Scaled and Non-Scaled Addressing Modes | | |
|---|---|---|
| example | Instruction syntax | operation |
| 7 | LDNDW<br>*++reg1[cst5], reg2 | pre-increment, scaled |
| 8 | LDNDW<br>*reg1++[cst5], reg2 | post-increment, scaled |

**[0105]** An advantage of scaled vs. non-scaled for the integer offset modes is that scaled provides a larger range of access whereas non-scaled provides finer granularity of access. Typically, when large offsets are used, they're multiples of the access size already. When small offsets are used, they're typically not, since typically a short moving distance is desired.

**[0106]** Scaled vs. non-scaled in register-offset modes is advantageous as well, but for different reasons. In scaled mode, the register offset usually corresponds to an array index of some sort. In non-scaled mode, the register offset may correspond to an image width or other stride parameter that isn't a multiple of the access width. For instance, accessing a 2 dimensional array whose row width is not a multiple of 8.

**[0107]** Figure 13 is a block diagram of an alternative embodiment of a digital system 1300 with processor core 1301 similar to CPU 10 of Figure 1. A direct mapped program cache 1710, having 16 Kbytes capacity in memory 1710b, is controlled by L1 Program (L1P) controller 1710a and connected thereby to the instruction fetch stage 10a. A 2-way set associative data cache 1720, having a 16 Kbyte capacity in memory 1720b, is controlled by L1 Data (L1D) controller 1720a and connected thereby to data units D1 and D2. An L2 memory 1730 having four banks of memory, 128 Kbytes total, is connected to L1P 1710a and to L1D 1720a to provide storage for data and programs. External memory interface (EMIF) 1750 provides a 64-bit data path to external memory, not shown, which provides memory data to L2 memory 1730 via extended direct memory access (DMA) controller 1740.

**[0108]** EMIF 1752 provides a 16-bit interface for access to external peripherals, not shown. Expansion bus 1770 provides host and I/O support similarly to host port 60/80 of Figure 1.

**[0109]** Three multi-channel buffered serial ports (McBSP) 1760, 1762, 1764 are connected to DMA controller 1740. A detailed description of a McBSP is provided in U.S. Patent S.N. 09/055,011 (TI-26204, Seshan, et al).

**[0110]** Advantageously, non-aligned accesses to a data cache 1720 is performed in the same amount of time as an aligned access to data cache 1720, as long as a miss does not occur. Likewise, advantageously, non-aligned accesses to a circular buffer region in data cache 1720 is performed in the same amount of time as an aligned access to a circular buffer region in data cache 1720, as long as a miss does not occur.

**[0111]** Figures 14A and 14B together is a block diagram of data cache L1D 1720b of Figure 13. L1D is a 16K byte 2-way associative cache that has 128 sets and a line size of 32 bytes. The data bus interface from L2 to L1D is 32 bytes wide (32 total, not 32 for A side and 32 for B side) - it takes one clock cycle to send a line from L2 to L1D. The store path from L1D to L2 is 16 bytes wide (16 total, not 16 for A side and 16 for B side), resulting in a 4 clock cycle line eviction. L1D operates solely as a cache and is not memory-mapped. Table 13 summarizes the features of cache L1D.

Table 13.

| L1D Cache Features | |
|---|---|
| Size | 16K bytes |
| Associativity | 2-way |
| Line Size | 32 bytes |
| L2 To L1D Load Bus Width | 32 bytes |
| L1D To L2 Store Bus Width | 16 bytes |
| Accesses Per Clock Cycle | 2 |
| Cycles For L1D Fill | 1 |
| Cycles For L1D Line Eviction | 4 |
| Interleave/Bank Size | 16 bit |

Table 13. (continued)

| L1D Cache Features | |
|---|---|
| Size | 16K bytes |
| Number Of Interleaves/Banks | 16 |
| Tag Array SRAM Size | 4 - 128 x 19 |
| Data Array SRAM Size | 16 - 512 x 16 |
| Write Hit Policy | Write back w/write-through tags |
| Write Miss Policy | no allocate |
| Replacement Strategy | LRU |

**[0112]** L1D must carry four status bits for each cache line, including a valid bit, a modified bit, an NW (no-write) bit, and an LRU bit. The valid bits, modified (dirty) bits, and NW may reside in the tag RAM. The LRU bits reside in registers for single-cycle read-modify-write operation.

**[0113]** Read misses in L1D cause the corresponding line to be brought into the cache. The line originates from L2 if it is contained there; otherwise, the I/O subsystem transfers the line to L2 from another memory-mapped location, and the line is forwarded to L1D. L1D must keep track of the modification state of each of its lines. If an L1D line has been modified, it must be written out to L2 when it is replaced (or when requested to do so by an L2 control register write).

**[0114]** L1D uses an LRU replacement strategy. A given line can reside at the same address offset within either way 0 or way 1 of the cache. Each line has an LRU bit that keeps track of which way was most recently used for the corresponding line. When a new line must replace a line that is already stored in the cache, it replaces the line that was least recently used and modifies the LRU status bit accordingly. If one of the two candidates for replacement is invalid, it is replaced without regard to LRU status, and the LRU bit indicates that way holding the new line is most recently used.

**[0115]** The L1D controller 1720a must not use an unitialized value of the LRU bit that is present following power-up/reset. Invalidate operations do not affect LRU status.

**[0116]** Read hits allow the DSP to continue execution without stalling. The LRU status might need be updated.

**[0117]** Write misses do not cause allocation in L1D; the write data is sent to a write buffer to await transfer to L2. Provided that the write buffer is not full, the DSP does not need to stall on write misses to L1D. The LRU status is unaffected by write misses.

**[0118]** On a write hit, data is written into the cache, and the DSP continues to execute without stalling. If the L1D line has not yet been modified, L1D must also perform a tag writethrough to L2, sending the tag value of the corresponding line to L2. L2 stores a copy of the tag for each line in L1D along with a dirty bit for the line, and it must be informed that the L1D line has been modified. With this mechanism L2 monitors the status of L1D data without the complexity or conflicts caused by snoop traffic. Tag writethrough should have a single cycle throughput, assuming no conflicts. The LRU status might need to be updated due to a write hit.

**[0119]** L1D adheres to the pipeline timing described in Table 14.

Table 14.

| Pipeline Description | | |
|---|---|---|
| Stage | Module | Action |
| E1 | DSP | Register file read and address generate |
| E2 | DSP | Send address and data (if a store) to L1D |
| E2 | L1D | Receive data and perform tag lookup to initiate an address compare |
| E3 | L1D | If address matches and tag is valid, initiate data RAM read or write. Else send data to write buffer for stores or stall DSP for loads |
| E4 | DSP | Receive load data from L1D |
| E4 | L1D | Send load data to DSP |
| E5 | DSP | Write load data into register file |

**[0120]** Figure 14a is a block diagram illustrating an implementation of the L1D tag array. Logically, 4 tag RAM's

1401-1404 are required for L1D. Both ways of the cache require a tag RAM, and the RAM's are replicated to allow both D units of the DSP to simultaneously perform tag lookups on different addresses. A first address is provided by .D1 on a first address bus 1410 and accesses tag RAMs 1401-1402. A second address is provided by .D2 on bus 1411 and accesses tag RAMs 1403-1404. When a line is replaced, the tag RAM's associated with both D units must be updated to maintain duplicate copies of the tags. A three-input mux 1420, 1421 is provided to allow tag comparisons to occur one cycle earlier.

**[0121]** The individual tag RAM's are 128 x 20. One bit of the 20 bits is used to store a physical address attribute map (PAAM) NW (no-write) bit. Aspects of a PAAM are described in detail in co-assigned U.S. Patent application Number 09/702,477 entitled System Address Properties Control Cache Memory, Including Physical Address Attribute Map (PAAM). The PAAM PC and NR bits do not need to be stored in L1D.

**[0122]** Figure 14B is a block diagram illustrating an implementation of the L1D data array 1720b. Note that the L1D Data diagram omits the logic for a write buffer. L1D contains a buffer of at least 58 bytes that resides between L1D and L2, and it is used for both victims and L1D write misses. The buffer can hold one 32 byte victim plus two L1D doubleword write misses, or it could hold up to ten L1D doubleword misses without a victim.

**[0123]** Each D unit of the DSP can access L1D 1720 with a load or store of a byte, halfword, word, or doubleword. When the two D units request a simultaneous access, there may or may not be stalls as a result of bank conflicts. Thirty-two bytes are available per cycle (an individual bank holds data for both ways of the cache). A doubleword access can occur at one of four offsets within a line, including 0, 8, 16, and 24 bytes. Assuming random behavior, there is a 25% chance of a conflict between two simultaneous doubleword accesses. Like doubleword requests, byte, half-word, and word requests may or may not have bank conflicts, but they are less likely to happen and easier to avoid from a programming standpoint.

**[0124]** Advantageously, L1D cache 1720b is not affected by a single non-aligned word or double word load or store transaction since a non-aligned transaction is presented to the cache as two aligned transfers. Byte swapping circuitry similar to Figure 10 and Figure 11 is included in controller block 1720a to provide alignment.

**[0125]** Figure 15 illustrates an exemplary implementation of a digital system that includes DSP 1 packaged in an integrated circuit 40 in a mobile telecommunications device, such as a wireless telephone 15. Wireless telephone 15 has integrated keyboard 12 and display 14. As shown in Figure 15, DSP 1 is connected to the keyboard 12, where appropriate via a keyboard adapter (not shown), to the display 14, where appropriate via a display adapter (not shown) and to radio frequency (RF) circuitry 16. The RF circuitry 16 is connected to an aerial 18. Advantageously, by allowing non-aligned accesses into linear regions or circular buffer regions in the memory subsystem of DSP 1, complex signal processing algorithms can be written in a more efficient manner to satisfy the demand for enhanced wireless telephony functionality. More importantly, non-aligned accesses into linear and circular buffer regions take the same amount of time as aligned access into the same regions, so that real time algorithms operate in a consistent, predictable manner.

**[0126]** Table 15 summarizes instruction operation and execution notations used throughout this document.

Table 15.

| Instruction Operation and Execution Notations | |
|---|---|
| Symbol | Meaning |
| long | 40-bit register value |
| +a | Perform twos-complement addition using the addressing mode defined by the AMR |
| -a | Perform twos-complement subtraction using the addressing mode defined by the AMR |
| xor | Bitwise exclusive OR |
| not | Bitwise logical complement |
| $b_{y..z}$ | Selection of bits y through z of bit string b |
| >>s | Shift right with sign extension |
| >>z | Shift right with a zero fill |
| x clear b,e | Clear a field in x, specified by b (beginning bit) and e (ending bit) |
| x exts l,r | Extract and sign-extend a field in x, specified by 1 (shift left value) and r (shift right value) |
| x extu l,r | Extract an unsigned field in x, specified by l (shift left value) and r (shift right value) |
| +s | Perform twos-complement addition and saturate the result to the result size, if an overflow or underflow occurs |

Table 15. (continued)

| Instruction Operation and Execution Notations | |
|---|---|
| Symbol | Meaning |
| -s | Perform twos-complement subtraction and saturate the result to the result size, if an overflow or underflow occurs |
| x set b,e | Set field in x, to all 1s specified by b (beginning bit) and e (ending bit) |
| lmb0(x) | Leftmost 0 bit search of x |
| lmb1(x) | Leftmost 1 bit search of x |
| norm(x) | Leftmost nonredundant sign bit of x |
| abs(x) | Absolute value of x |
| and | Bitwise AND |
| bi | Select bit i of source/destination b |
| bit_cou nt | Count the number of bits that are 1 in a specified byte |
| bit_rev erse | Reverse the order of bits in a 32-bit register |
| byte0 | 8-bit value in the least significant byte position in 32-bit register (bits 0-7) |
| byte1 | 8-bit value in the next to least significant byte position in 32-bit register (bits 8-15) |
| byte2 | 8-bit value in the next to most significant byte position in 32-bit register (bits 16-23) |
| byte3 | 8-bit value in the most significant byte position in 32-bit register (bits 24-31) |
| bv2 | Bit Vector of two flags for s2 or u2 data type |
| bv4 | Bit Vector of four flags for s4 or u4 data type |
| cond | Check for either creg equal to 0 or creg not equal to 0 |
| creg | 3-bit field specifying a conditional register |
| cstn | n-bit constant field (for example, cst5) |
| dst_h or dst_o | msb32 of dst (placed in odd register of 64-bit register pair) |
| dst_l or dst_e | lsb32 of dst (place in even register of a 64-bit register pair) |
| dws4 | Four packed signed 16-bit integers in a 64-bit register pair |
| dwu4 | Four packed unsigned 16-bit integers in a 64-bit register pair |
| gmpy | Galois Field Multiply |
| i2 | Two packed 16-bit integers in a single 32-bit register |
| i4 | Four packed 8-bit integers in a single 32-bit register |
| int | 32-bit integer value |
| lsbn or LSBn | n least significant bits (for example, lsbl6) |
| msbn or MSBn | n most significant bits (for example, msb16) |
| nop | No operation |
| or | Bitwise OR |
| R | Any general-purpose register |
| rotl | Rotate left |
| sat | Saturate |
| sbyte0 | Signed 8-bit value in the least significant byte position in 32-bit register (bits 0-7) |
| sbyte1 | Signed 8-bit value in the next to least significant byte position in 32-bit register (bits 8-15) |

Table 15. (continued)

| Instruction Operation and Execution Notations | |
|---|---|
| Symbol | Meaning |
| sbyte2 | Signed 8-bit value in the next to most significant byte position in 32-bit register (bits 16-23) |
| sbyte3 | Signed 8-bit value in the most significant byte position in 32-bit register (bits 24-31) |
| scstn | Signed n-bit constant field (for example, scst7) |
| se | Sign-extend |
| sint | Signed 32-bit integer value |
| slsb16 | Signed 16-bit integer value in lower half of 32-bit register |
| smsb16 | Signed 16-bit integer value in upper half of 32-bit register |
| s2 | Two packed signed 16-bit integers in a single 32-bit register |
| s4 | Four packed signed 8-bit integers in a single 32-bit register |
| slong | Signed 64-bit integer value |
| ubyte0 | Unsigned 8-bit value in the least significant byte position in 32-bit register (bits 0-7) |
| ubyte1 | Unsigned 8-bit value in the next to least significant byte position in 32-bit register (bits 8-15) |
| ubyte2 | Unsigned 8-bit value in the next to most significant byte position in 32-bit register (bits 16-23) |
| ubyte3 | Unsigned 8-bit value in the most significant byte position in 32-bit register (bits 24-31) |
| ucstn | n-bit unsigned constant field (for example, ucst5) |
| uint | Unsigned 32-bit integer value |
| ulong | Unsigned 64-bit integer value |
| ulsb16 | Unsigned 16-bit integer value in lower half of 32-bit register |
| umsb16 | Unsigned 16-bit integer value in upper half of 32-bit register |
| u2 | Two packed unsigned 16-bit integers in a single 32-bit register |
| u4 | Four packed unsigned 8-bit integers in a single 32-bit register |
| xi2 | Two packed 16-bit integers in a single 32-bit register that can optionally use cross path |
| xi4 | Four packed 8-bit integers in a single 32-bit register that can optionally use cross path |
| xsint | Signed 32-bit integer value that can optionally use cross path |
| xs2 | Two packed signed 16-bit integers in a single 32-bit register that can optionally use cross path |
| xs4 | Four packed signed 8-bit integers in a single 32-bit register that can optionally use cross path |
| xuint | Unsigned 32-bit integer value that can optionally use cross path |
| xu2 | Two packed unsigned 16-bit integers in a single 32-bit register that can optionally use cross path |
| xu4 | Four packed unsigned 8-bit integers in a single 32-bit register that can optionally use cross path |
| → | Assignment |
| + | Addition |
| ++ | Increment by one |
| x | Multiplication |
| - | Subtraction |
| > | Greater than |
| < | Less than |
| << | Shift left |

Table 15. (continued)

| Instruction Operation and Execution Notations | |
|---|---|
| Symbol | Meaning |
| >> | Shift right |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| == | Equal to |
| ~ | Logical Inverse |
| & | Logical And |

**[0127]** Fabrication of digital system 1 and 1300 involves multiple steps of implanting various amounts of impurities into a semiconductor substrate and diffusing the impurities to selected depths within the substrate to form transistor devices. Masks are formed to control the placement of the impurities. Multiple layers of conductive material and insulative material are deposited and etched to interconnect the various devices. These steps are performed in a clean room environment.

**[0128]** A significant portion of the cost of producing the data processing device involves testing. While in wafer form, individual devices are biased to an operational state and probe tested for basic operational functionality. The wafer is then separated into individual dice which may be sold as bare die or packaged. After packaging, finished parts are biased into an operational state and tested for operational functionality.

**[0129]** Thus, a digital system is provided with a processor having an improved instruction set architecture. The .D units can access words and double words on any byte boundary by using non-aligned load and store instructions, maintain the same instruction execution timing for aligned and non-aligned memory accesses. Advantageously, a significant amount of additional hardware is not required to perform the non-aligned accesses because a single non-aligned access can be performed using the resources of two separate aligned access ports.

**[0130]** As used herein, the terms "applied," "connected," and "connection" mean electrically connected, including where additional elements may be in the electrical connection path. "Associated" means a controlling relationship, such as a memory resource that is controlled by an associated port. The terms assert, assertion, de-assert, de-assertion, negate and negation are used to avoid confusion when dealing with a mixture of active high and active low signals. Assert and assertion are used to indicate that a signal is rendered active, or logically true. De-assert, de-assertion, negate, and negation are used to indicate that a signal is rendered inactive, or logically false.

**[0131]** While the invention has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various other embodiments of the invention will be apparent to persons skilled in the art upon reference to this description. For example, more than two target memory ports may be provided. Different data widths may be provided, such as 128-bit data items, for example. As long as the size of a non-aligned data item is less than or equal to the size of each aligned access port, then two access ports can be shared to provide a single non-aligned access without adding significant additional resources.

**[0132]** Scaling circuitry may be included or not included within the address generation circuitry. Scaling/non-scaling can be selectively included in instructions for data sizes other than double words.

**[0133]** Circular buffer address circuitry may be included or not included within the address generation circuitry.

**[0134]** In the digital system according to the claims, the first instruction type is a non-aligned access type, and wherein the second instruction type is an aligned access type.

**[0135]** In the digital system according to the claims, the second load/store unit is operable to execute a non-memory access instruction in parallel with the first load/store unit accessing the memory subsystem for an instruction of the first type.

**[0136]** In the digital system according to the claims, the memory subsystem is a cache memory.

**[0137]** In the digital system according to the claims, the microprocessor is a digital signal processor.

**[0138]** In the digital system according to the claims, the address circuitry comprises: combination circuitry connected to receive a base address value and an offset value, operable to combine the base address value and the offset value to form a base-offset address, wherein the base-offset address is selectively coupled to the first address output; and adder circuitry connected to receive the base-offset address and a line size value, operable to add the line size value to the base-offset address to form an augmented address, wherein the augmented address is selectively coupled the second address output.

**[0139]** In the digital system according to the claims, the step of extracting loads a data value from the non-aligned data item into the microprocessor.

**[0140]** In the digital system according to the claims, the step of extracting stores a data value in the non-aligned data item in the memory subsystem by storing a first portion of the non-aligned data item in a first aligned data item and storing a second portion of the non-aligned data item in a second aligned data item.

**[0141]** In the digital system according to the claims, the first data item is an aligned data item and wherein the second data item is an aligned data item.

**[0142]** A digital system embodying the present invention comprises: a microprocessor having at least a first load/store unit; a memory subsystem having at least first memory port connected to the first load/store unit; address generation circuitry in the first load/store unit having a first address output connected to the first memory port, the address generation circuitry operable to provide a first byte address on the first address output; and an extraction circuit connected to the first memory port, wherein the extraction circuit is operable to provide a first non-aligned multi-byte data item to the first load/store unit responsive to the first byte address.

## Claims

1. A digital system, comprising;

   a microprocessor having at least a first load/store unit and a second load/store unit;
   a memory subsystem having at least a first memory port connected to the first load/store unit and a second memory port connected to the second load/store unit;
   address generation circuitry in the first load/store unit having a first address output connected to the first memory port and a second address output selectively connected to the second memory port, the address generation circuitry operable to provide a first address on the first address output and a second address on the second address output; and
   an extraction circuit connected to the first memory port, wherein the extraction circuit is operable to provide a first non-aligned data item to the first load/store unit extracted from a first data item accessed in response to the first address and from a second data item accessed in response to the second address.

2. The digital system of Claim 1, further comprising insertion circuitry connected to the first memory port, wherein the insertion circuitry is operable to receive a second non-aligned data item from the first load/store unit and to store a first portion of the second non-aligned data item in a first data item in the memory subsystem responsive to the first address and to store a second portion of the second non-aligned data item in a second data item in the memory subsystem responsive to the second address.

3. The digital system according to any preceding Claim, wherein the address generation circuitry is operable to provide the first address and second address to the memory subsystem to access the memory subsystem in response to a first instruction type and to provide only the first address to the memory subsystem to access the memory subsystem in response to a second instruction type.

4. The digital system according to any preceding Claim, wherein the second load/store unit comprises address generation circuitry with a first address output selectively connected to the second memory port, such that the second load/store unit is operable to transfer a data item to the second memory port in parallel with the first load/store unit transferring a data item to the first memory port.

5. The digital system according to any preceding Claim, wherein the address generation circuitry of the second load/store unit is operable to provide the first address on a second address output selectively connected to the first memory port and the second address on the first address output for accessing the memory subsystem for an instruction of the first type.

6. The digital system according to any preceding Claim being a cellular telephone, further comprising:

   an integrated keyboard connected to the CPU via a keyboard adapter;
   a display, connected to the CPU via a display adapter;
   radio frequency (RF) circuitry connected to the CPU; and
   an aerial connected to the RF circuitry.

7. The digital system according to any preceding Claim, wherein the memory subsystem comprises:

a plurality of memory banks connected to the extraction circuitry;
decode circuitry connected to the first memory port and to the second memory port; and
a plurality of address multiplexers connected respectively to the plurality of memory banks with first input of each of the plurality of multiplexers connected a receive an address from the first memory port and a second input connected to receive an address from the second memory port, each of the plurality of address multiplexers having a select control separately connected to the decode circuitry, such that the decode circuitry is operable to individually control each of the plurality of address multiplexers.

**8.** A method of operating a microprocessor, comprising the steps of:

fetching a first instruction for execution, wherein the first instruction is a non-aligned access type instruction and wherein the first instruction references a non-aligned data item in a memory subsystem region;
decoding the instruction to form a plurality of fields;
forming a first address and accessing a first data item via a first port of the memory subsystem;
forming a second address and accessing a second data item via a second port of the memory subsystem, such that the first address and second address are formed in a simultaneous manner and such that the first data item and the second data item are accessed in a simultaneous manner: and
extracting the non-aligned data item from the first data item and the second data item.

**9.** The method of Claim 8, wherein the step of forming a first address comprises the step of combining a base address value and an offset value in accordance with one of the plurality of fields of the instruction.

**10.** The method according to any of Claims 8-9, wherein the step of forming a second address comprises the step of adding a line size value in accordance with another one of the fields of the instruction.

**11.** The method according to any of Claims 8-10, further comprising the steps of:

fetching a second instruction and a third instruction for parallel execution, wherein the second instruction and the third instruction are both aligned access type instructions;
forming a third address in accordance with the second instruction and accessing a third data item via the first port of the memory subsystem;
forming a fourth address in accordance with the third instruction and accessing a fourth data item via the second port of the memory subsystem, such that the third address and the fourth address are formed in a simultaneous manner and such that the third data item and the fourth data item are accessed in a simultaneous manner.

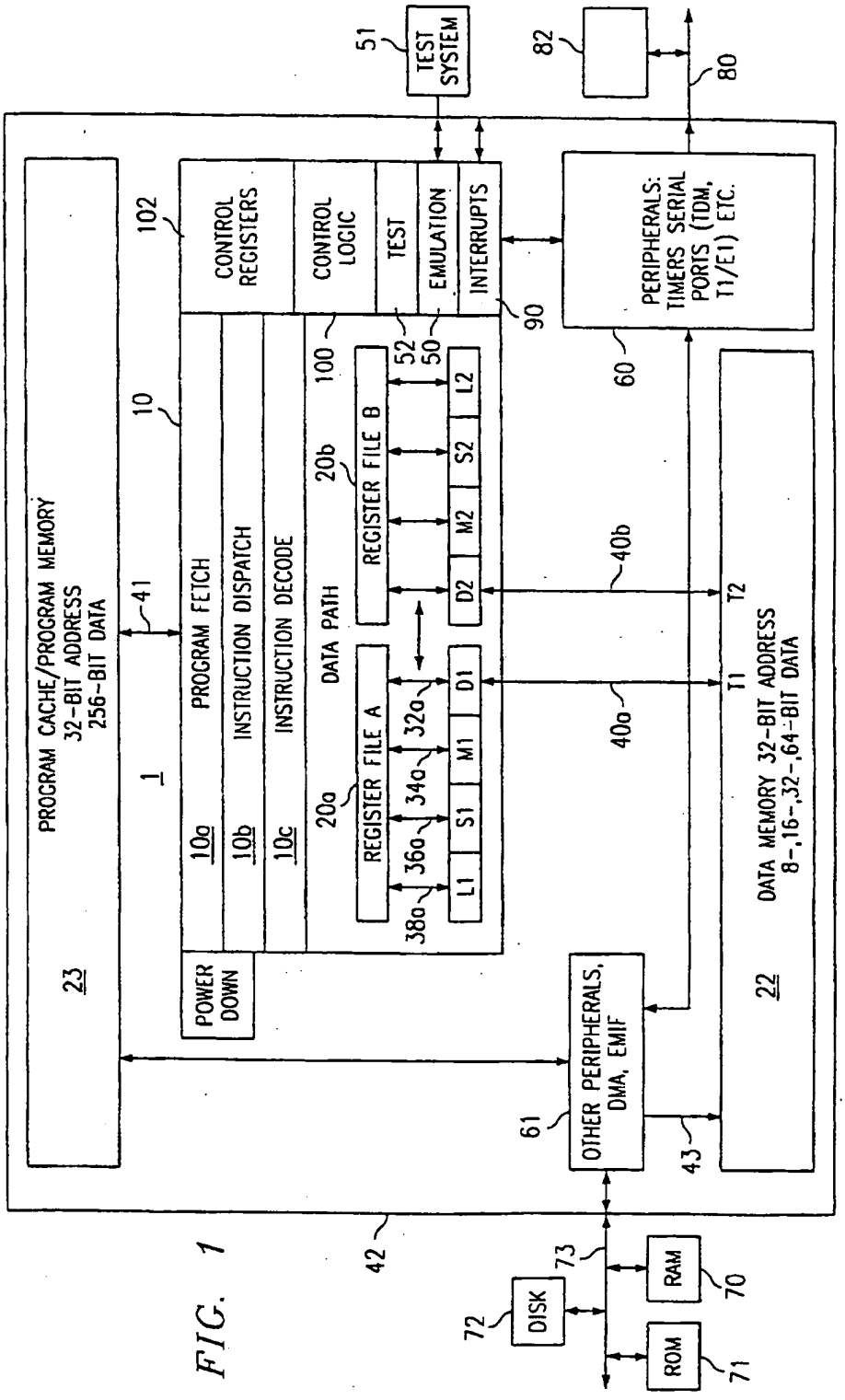

PROGRAM CACHE/PROGRAM MEMORY
32-BIT ADDRESS
256-BIT DATA
23
1
10
102
CONTROL REGISTERS
CONTROL LOGIC
TEST
EMULATION
INTERRUPTS
51
TEST SYSTEM
82
80
90
PERIPHERALS: TIMERS SERIAL PORTS (TDM, T1/E1) ETC.
60
52
50
100
10a
PROGRAM FETCH
10b
INSTRUCTION DISPATCH
10c
INSTRUCTION DECODE
20b
REGISTER FILE B
L2
S2
M2
D2
41
20a
DATA PATH
REGISTER FILE A
L1
S1
M1
D1
32a
34a
36a
38a
40b
40a
T2
T1
DATA MEMORY 32-BIT ADDRESS
8-,16-,32-,64-BIT DATA
22
POWER DOWN
OTHER PERIPHERALS, DMA, EMIF
61
43
42
73
DISK
72
RAM
70
ROM
71

FIG. 1

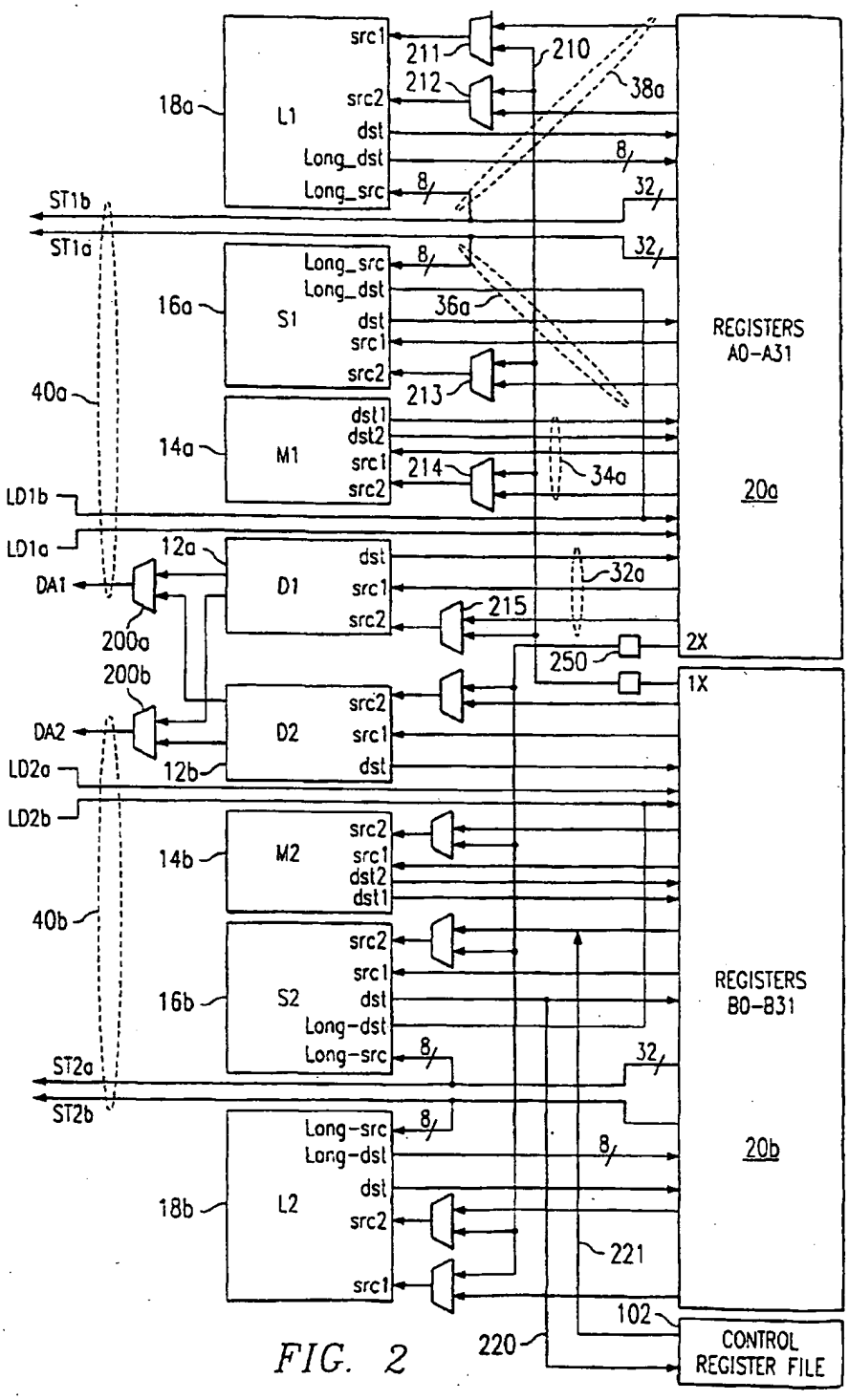

18a
L1
src1
src2
dst
Long_dst
Long_src
211
212
210
38a
8
32
ST1b
ST1a
16a
S1
Long_src
Long_dst
dst
src1
src2
36a
213
40a
14a
M1
dst1
dst2
src1
src2
214
34a
REGISTERS
A0-A31
20a
LD1b
LD1a
DA1
12a
D1
dst
src1
src2
32a
215
200a
200b
250
2X
1X
DA2
D2
12b
LD2a
LD2b
14b
M2
dst2
dst1
40b
16b
S2
Long-dst
Long-src
REGISTERS
B0-B31
ST2a
ST2b
18b
L2
20b
221
102
CONTROL
REGISTER FILE
220

*FIG. 2*

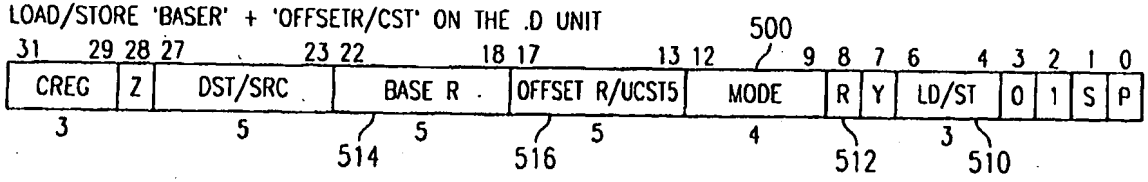



*FIG. 3A*

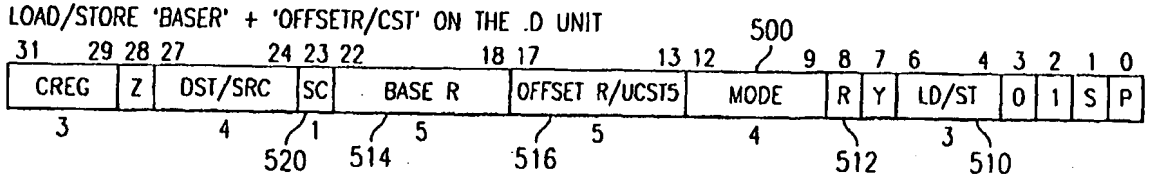



*FIG. 3B*

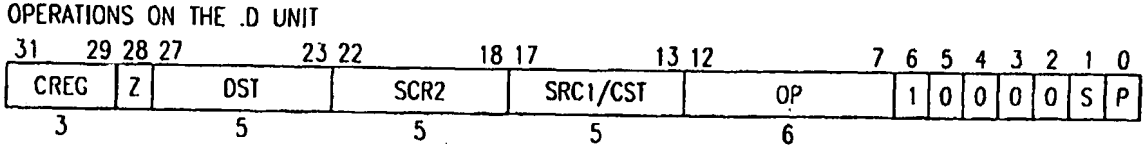



*FIG. 3C*

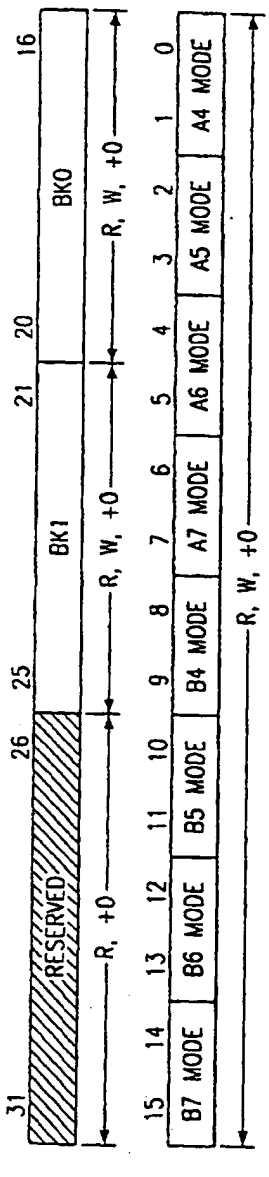

31
26
25
21
20
16
RESERVED
BK1
BK0
R, +0
R, W, +0
R, W, +0
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
B7 MODE
B6 MODE
B5 MODE
B4 MODE
A7 MODE
A6 MODE
A5 MODE
A4 MODE
R, W, +0
NOTE: R=READABLE BY THE MVC INSTRUCTION
W=WRITEABLE BY THE MVC INSTRUCTION
+0=VALUE IS ZERO AFTER RESET

FIG. 4

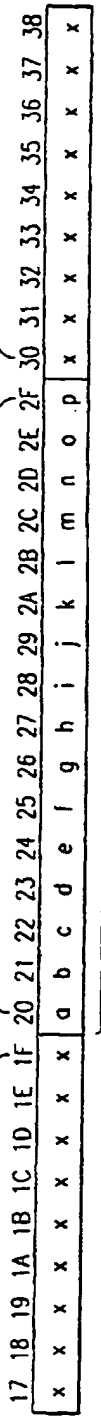

17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38
x x x x x x x x a b c d e f g h i j k l m n o p x x x x x x x x x x
506
502
504
508
500

FIG. 5A

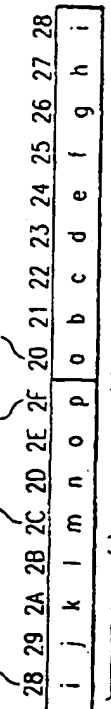



FIG. 5B

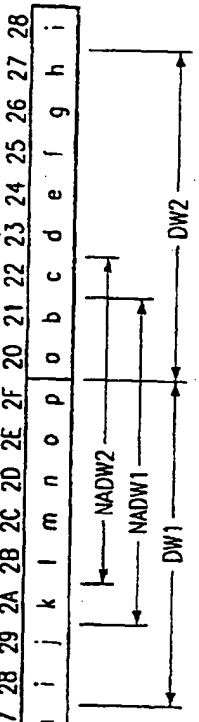



FIG. 5C




FIG. 6

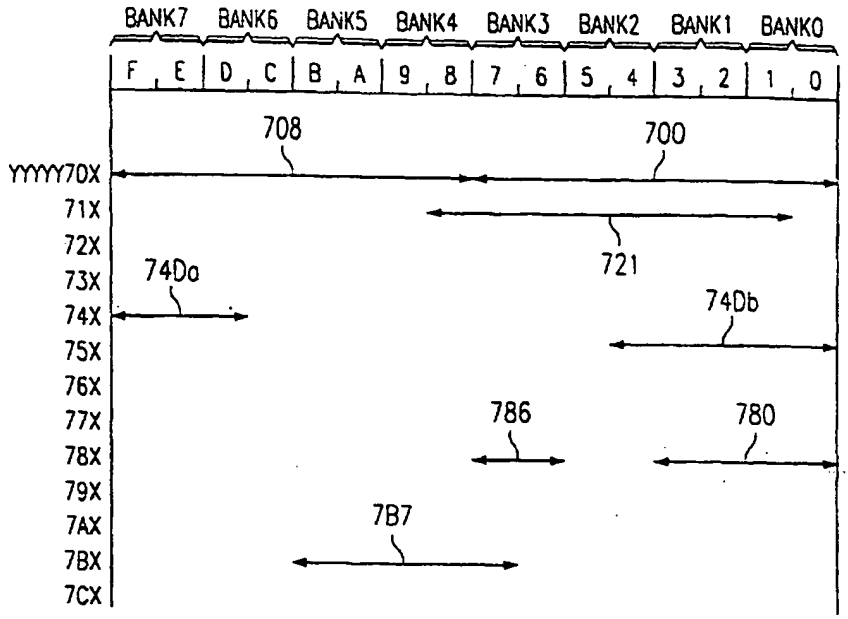

BANK7
BANK6
BANK5
BANK4
BANK3
BANK2
BANK1
BANK0
F E D C B A 9 8 7 6 5 4 3 2 1 0
708
700
YYYY70X
71X
72X
73X
74X
75X
76X
77X
78X
79X
7AX
7BX
7CX
721
74Da
74Db
786
780
7B7

FIG. 7

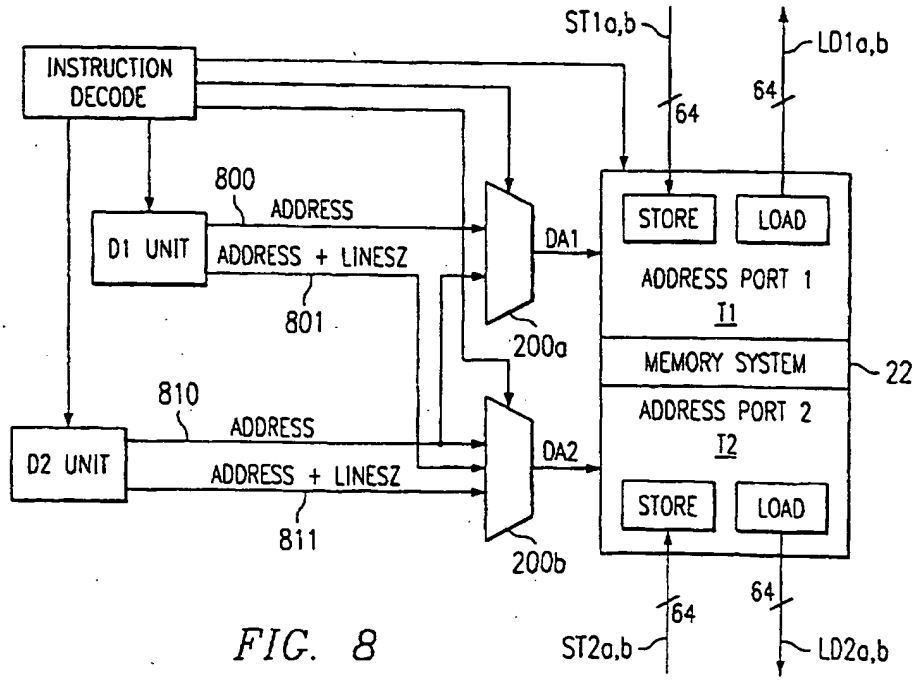

INSTRUCTION DECODE
D1 UNIT
D2 UNIT
800
ADDRESS
ADDRESS + LINESZ
801
810
ADDRESS
ADDRESS + LINESZ
811
DA1
DA2
200a
200b
ST1a,b
LD1a,b
64
64
STORE
LOAD
ADDRESS PORT 1
I1
MEMORY SYSTEM
22
ADDRESS PORT 2
I2
STORE
LOAD
64
64
ST2a,b
LD2a,b

FIG. 8

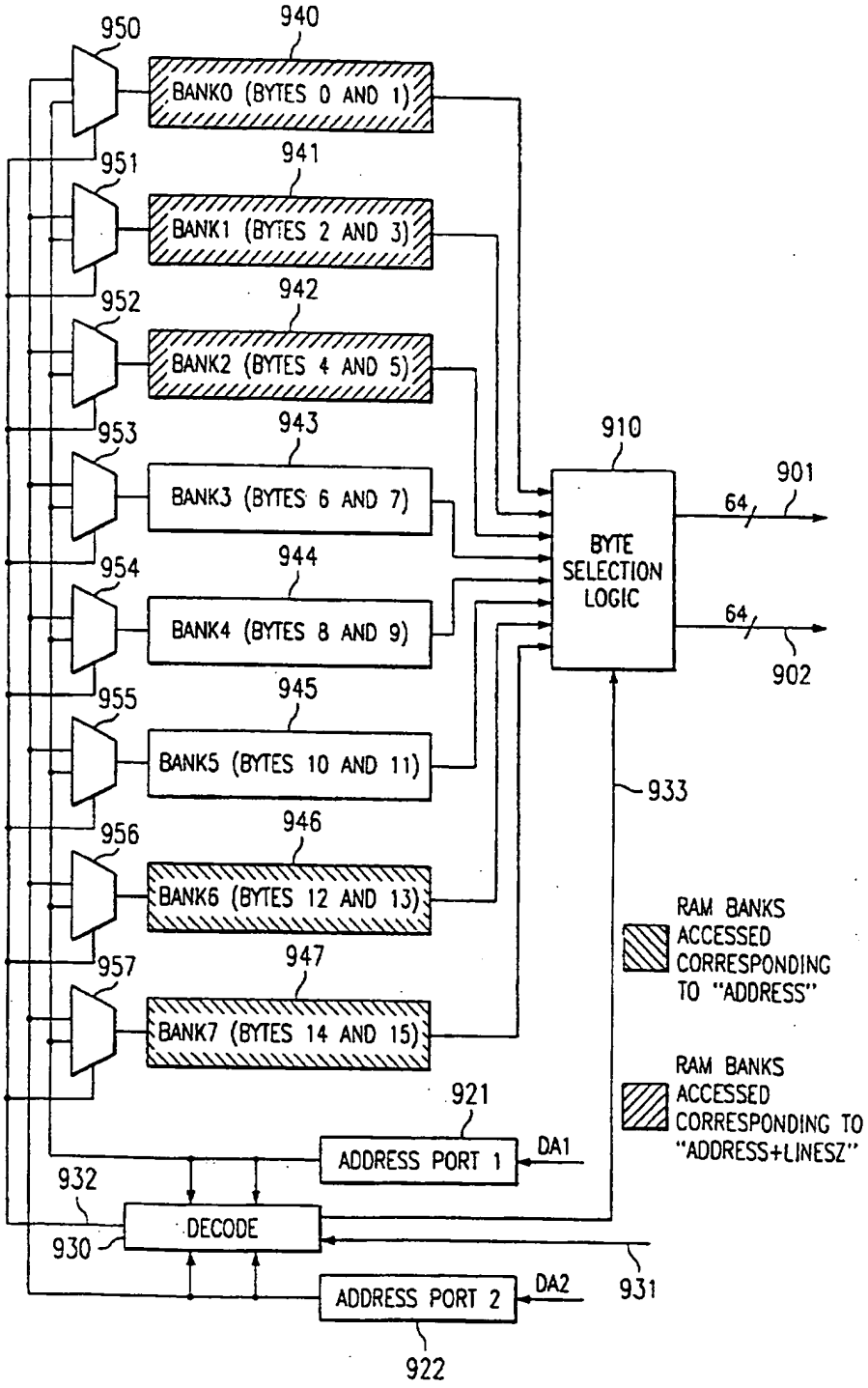

950
940
BANK0 (BYTES 0 AND 1)
951
941
BANK1 (BYTES 2 AND 3)
952
942
BANK2 (BYTES 4 AND 5)
953
943
BANK3 (BYTES 6 AND 7)
954
944
BANK4 (BYTES 8 AND 9)
955
945
BANK5 (BYTES 10 AND 11)
956
946
BANK6 (BYTES 12 AND 13)
957
947
BANK7 (BYTES 14 AND 15)
910
BYTE SELECTION LOGIC
901
64
64
902
933
RAM BANKS ACCESSED CORRESPONDING TO "ADDRESS"
RAM BANKS ACCESSED CORRESPONDING TO "ADDRESS+LINESZ"
921
ADDRESS PORT 1
DA1
932
DECODE
930
931
ADDRESS PORT 2
DA2
922

*FIG. 9*

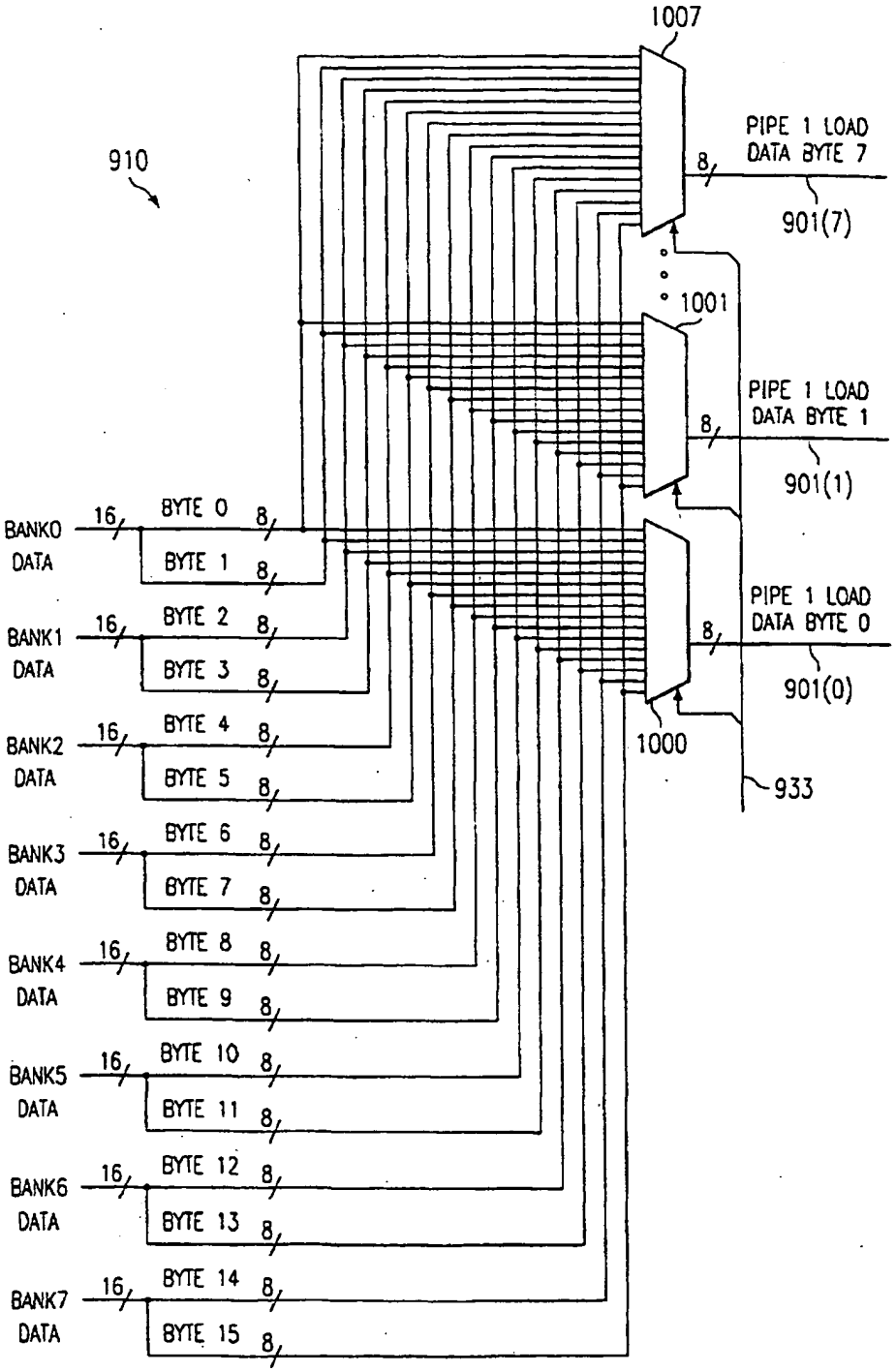

910
1007
1001
1000
933
PIPE 1 LOAD DATA BYTE 7
PIPE 1 LOAD DATA BYTE 1
PIPE 1 LOAD DATA BYTE 0
901(7)
901(1)
901(0)
8
BANK0 DATA
BANK1 DATA
BANK2 DATA
BANK3 DATA
BANK4 DATA
BANK5 DATA
BANK6 DATA
BANK7 DATA
16
BYTE 0
BYTE 1
BYTE 2
BYTE 3
BYTE 4
BYTE 5
BYTE 6
BYTE 7
BYTE 8
BYTE 9
BYTE 10
BYTE 11
BYTE 12
BYTE 13
BYTE 14
BYTE 15

*FIG. 10*

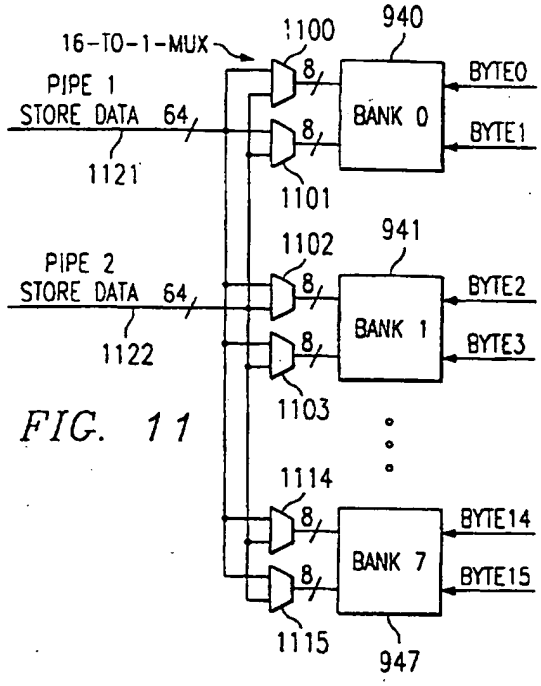



FIG. 11

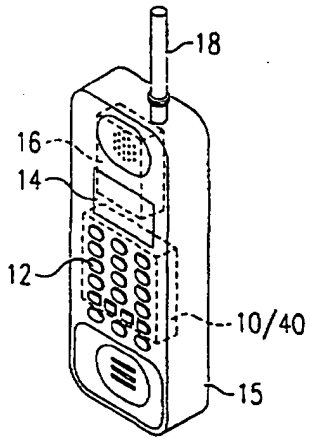



FIG. 15

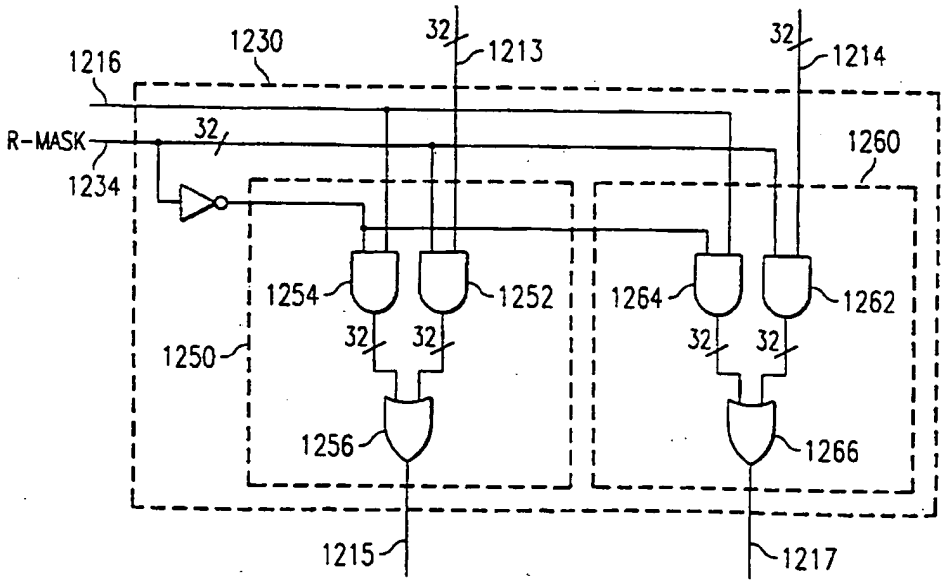



FIG. 12B

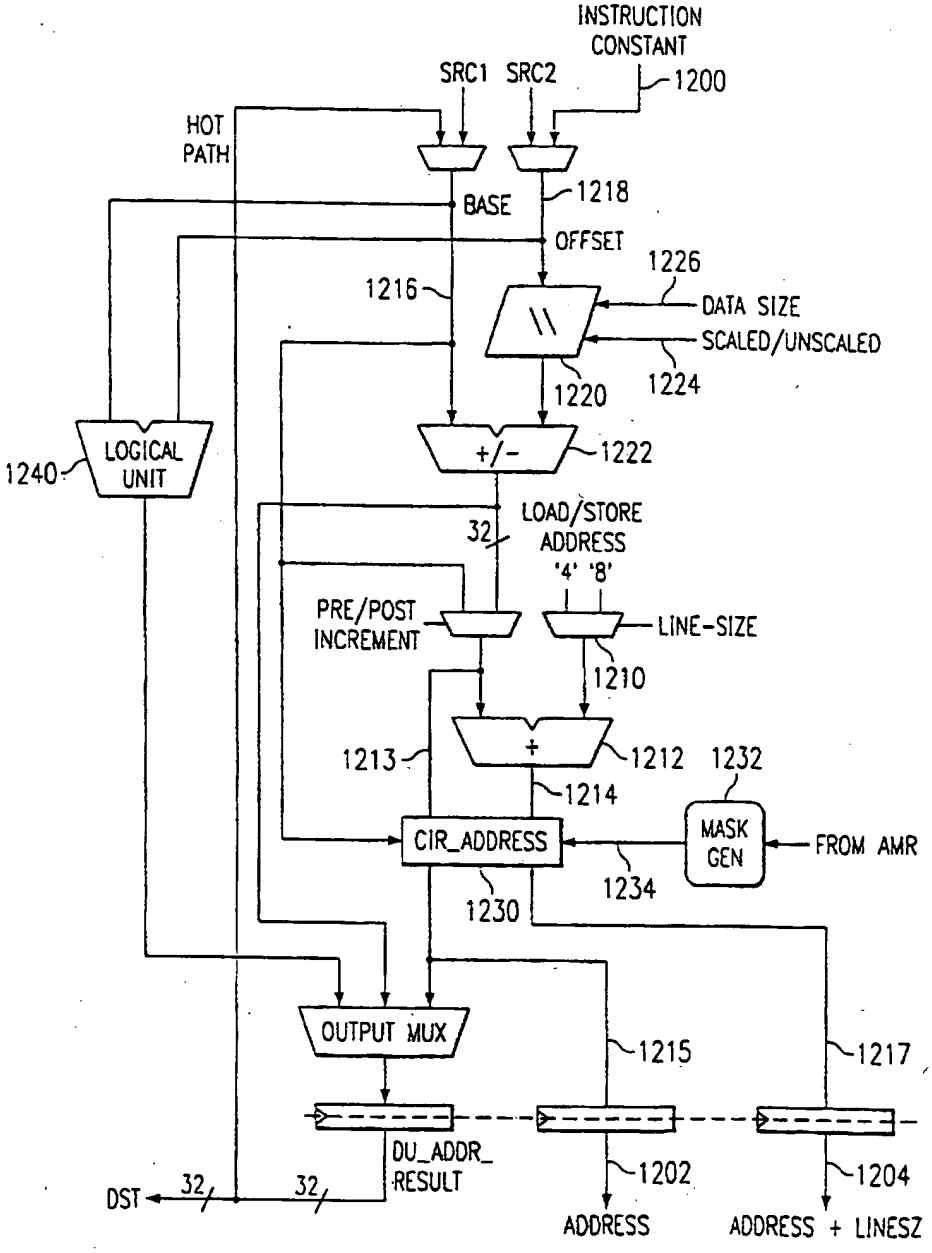

INSTRUCTION CONSTANT
SRC1
SRC2
1200
HOT PATH
BASE
1218
OFFSET
1226
1216
DATA SIZE
SCALED/UNSCALED
1224
1220
1240
LOGICAL UNIT
+/-
1222
32
LOAD/STORE ADDRESS
'4' '8'
PRE/POST INCREMENT
LINE-SIZE
1210
1213
+
1212
1232
1214
MASK GEN
FROM AMR
CIR_ADDRESS
1234
1230
OUTPUT MUX
1215
1217
DU_ADDR_ RESULT
1202
1204
DST
32
32
ADDRESS
ADDRESS + LINESZ

*FIG. 12A*

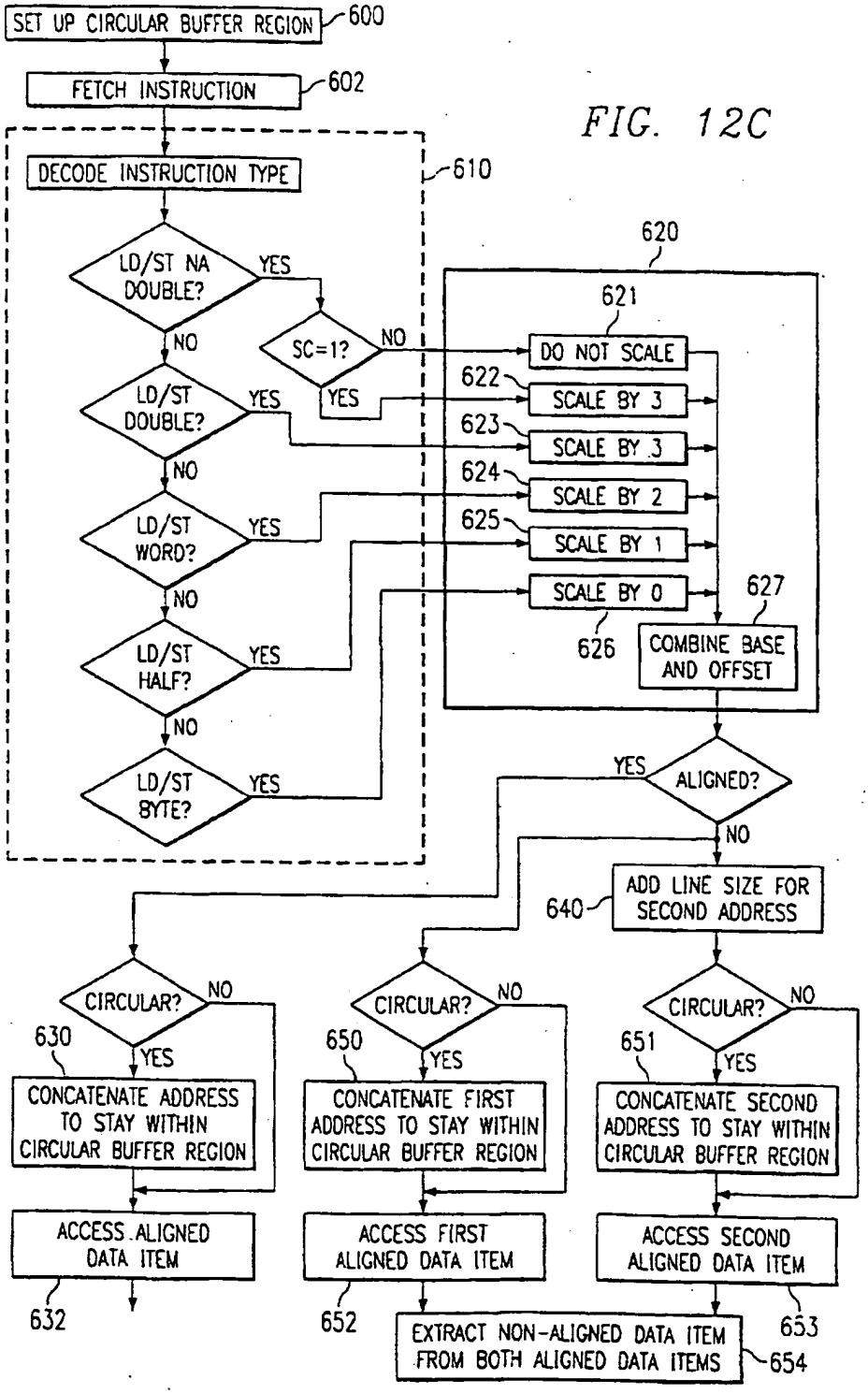

SET UP CIRCULAR BUFFER REGION
600
FETCH INSTRUCTION
602
DECODE INSTRUCTION TYPE
610
LD/ST NA DOUBLE?
YES
NO
SC=1?
NO
YES
LD/ST DOUBLE?
YES
NO
LD/ST WORD?
YES
NO
LD/ST HALF?
YES
NO
LD/ST BYTE?
YES
620
621
DO NOT SCALE
622
SCALE BY 3
623
SCALE BY 3
624
SCALE BY 2
625
SCALE BY 1
SCALE BY 0
626
627
COMBINE BASE AND OFFSET
ALIGNED?
YES
NO
640
ADD LINE SIZE FOR SECOND ADDRESS
CIRCULAR?
NO
YES
630
CONCATENATE ADDRESS TO STAY WITHIN CIRCULAR BUFFER REGION
ACCESS ALIGNED DATA ITEM
632
CIRCULAR?
NO
YES
650
CONCATENATE FIRST ADDRESS TO STAY WITHIN CIRCULAR BUFFER REGION
ACCESS FIRST ALIGNED DATA ITEM
652
CIRCULAR?
NO
YES
651
CONCATENATE SECOND ADDRESS TO STAY WITHIN CIRCULAR BUFFER REGION
ACCESS SECOND ALIGNED DATA ITEM
653
EXTRACT NON-ALIGNED DATA ITEM FROM BOTH ALIGNED DATA ITEMS
654

FIG. 12C

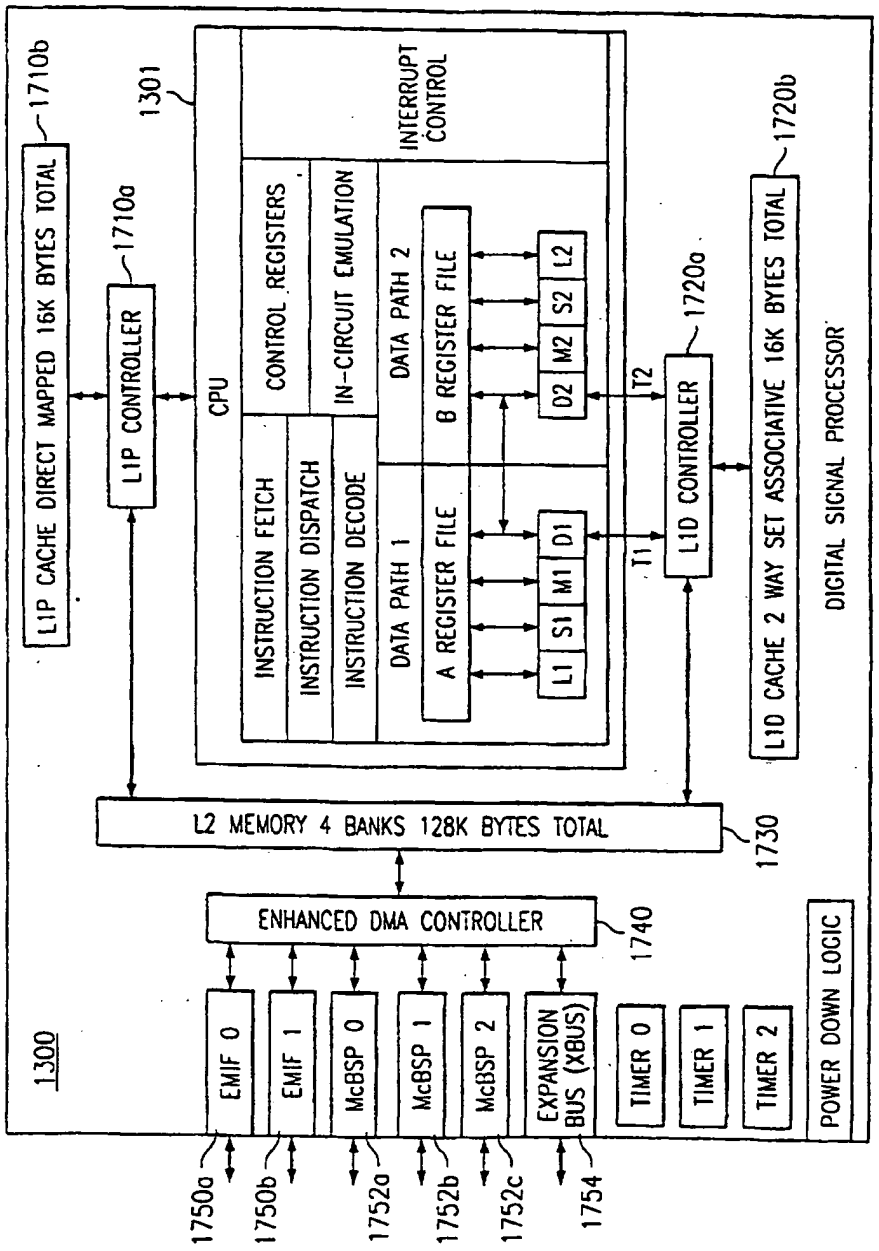

1300
L1P CACHE DIRECT MAPPED 16K BYTES TOTAL
1710b
L1P CONTROLLER
1710a
1301
CPU
INSTRUCTION FETCH
INSTRUCTION DISPATCH
INSTRUCTION DECODE
DATA PATH 1
A REGISTER FILE
L1
S1
M1
D1
CONTROL REGISTERS
IN-CIRCUIT EMULATION
DATA PATH 2
B REGISTER FILE
D2
M2
S2
L2
INTERRUPT CONTROL
T1
T2
L1D CONTROLLER
1720a
L1D CACHE 2 WAY SET ASSOCIATIVE 16K BYTES TOTAL
1720b
DIGITAL SIGNAL PROCESSOR
L2 MEMORY 4 BANKS 128K BYTES TOTAL
1730
ENHANCED DMA CONTROLLER
1740
EMIF 0
EMIF 1
McBSP 0
McBSP 1
McBSP 2
EXPANSION BUS (XBUS)
TIMER 0
TIMER 1
TIMER 2
POWER DOWN LOGIC
1750a
1750b
1752a
1752b
1752c
1754

*FIG. 13*

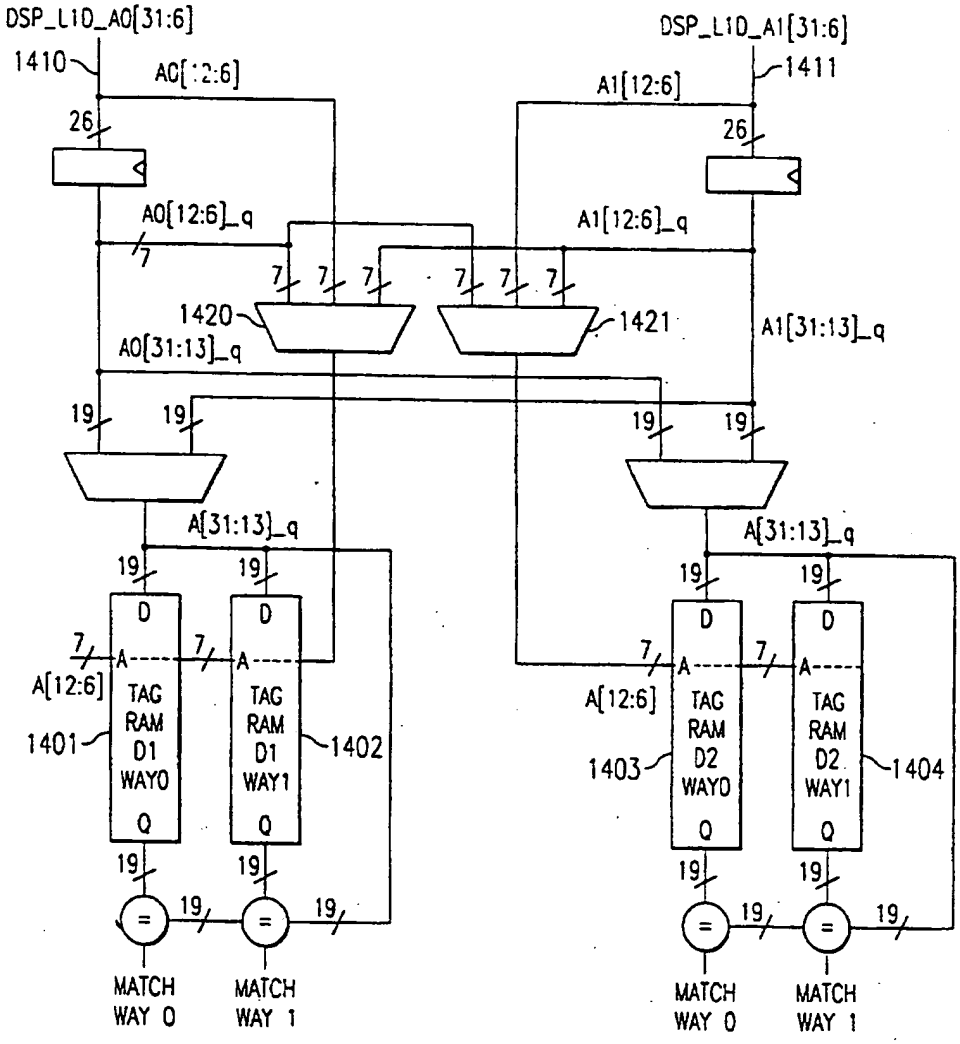
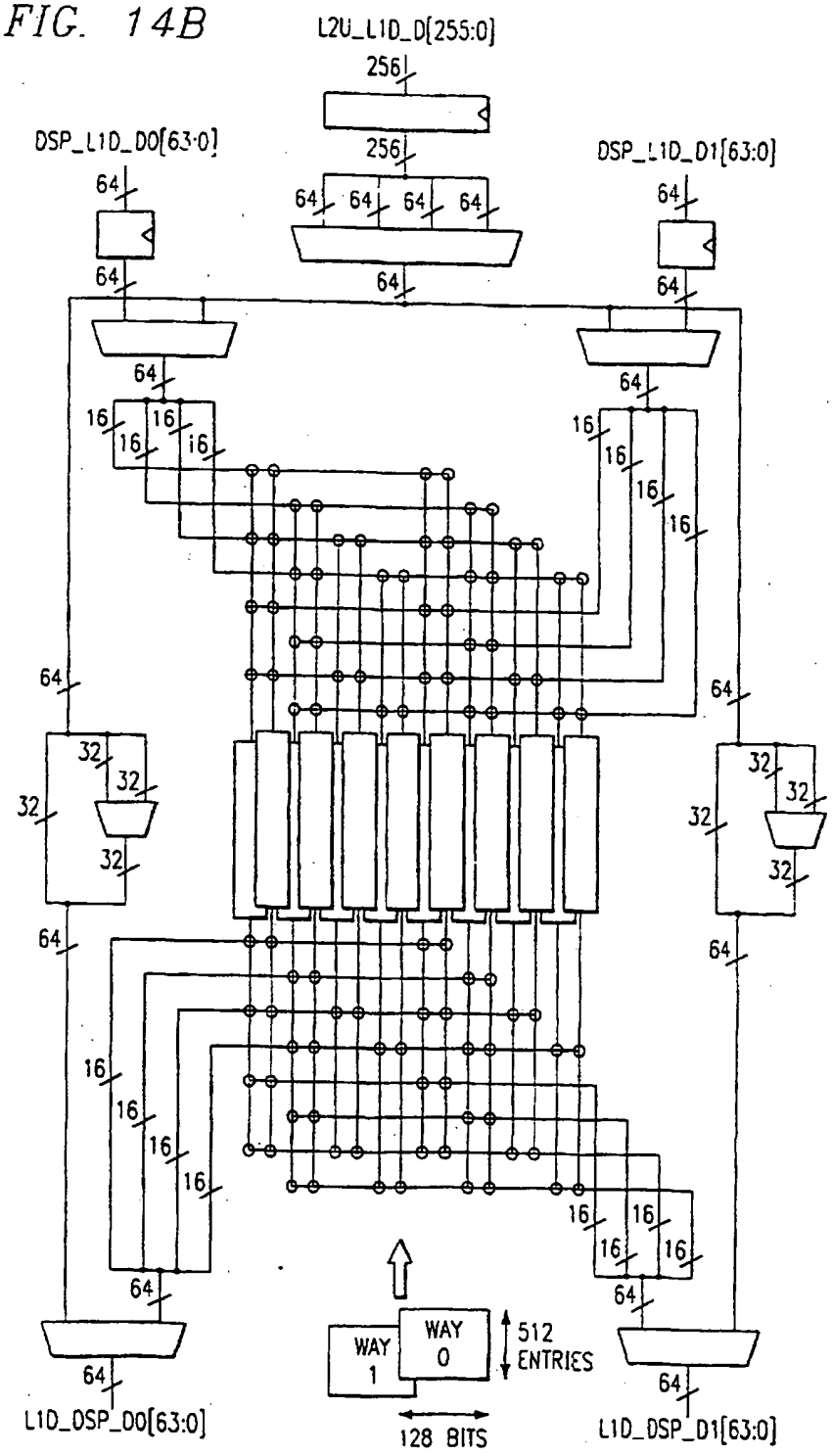

DSP_L1D_A0[31:6]
1410
A0[31:6]
26
A0[12:6]_q
7
1420
A0[31:13]_q
19
19
A[31:13]_q
19
19
D
A
7
A[12:6]
TAG
RAM
D1
WAY0
Q
1401
19
=
MATCH
WAY 0
D
A
7
TAG
RAM
D1
WAY1
Q
1402
19
=
MATCH
WAY 1
19
DSP_L1D_A1[31:6]
1411
A1[12:6]
26
A1[12:6]_q
7
1421
A1[31:13]_q
19
19
A[31:13]_q
19
19
D
A
7
A[12:6]
TAG
RAM
D2
WAY0
Q
1403
19
=
MATCH
WAY 0
D
A
7
TAG
RAM
D2
WAY1
Q
1404
19
=
MATCH
WAY 1
19

*FIG. 14A*

FIG. 14B


L2U_L1D_D[255:0]
256
256
DSP_L1D_D0[63:0]
DSP_L1D_D1[63:0]
64
64 64 64 64
64
64
64
64
64
64
16
16
16
16
16
16
16
16
16
16
16
16
16
16
16
16
64
64
32
32
32
32
32
32
64
64
64
64
16
16
16
16
16
16
16
16
64
64
WAY 1
WAY 0
512 ENTRIES
128 BITS
64
64
L1D_DSP_D0[63:0]
L1D_DSP_D1[63:0]